TURING 图灵程序设计丛书

涵盖最新版本1.2.x

ng-book

The Complete Book on AngularJS

AngularJS权威教程

[美] Ari Lerner 著

赵望野 徐飞 何鹏飞 译

人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

作 者 简 介



Ari Lerner

全栈工程师，拥有多年AngularJS经验，自办并运营AngularJS电子报ng-newsletter.com，在著名硅谷工程师培训学校Hack Reactor担任AngularJS讲师。他的工作涉及软件开发的各个层次，包括基础设施开发、前端应用开发和性能优化。他目前住在旧金山一个阳光明媚的地方，还是FullStack.io创始人。

译 者 简 介



赵望野

前端工程师，前端基础技术组leader，曾经负责豌豆荚2.0的前端架构设计和主要开发工作，目前负责Front-end Technical Infrastructure的建设，在工作中有丰富的AngularJS使用经验。新浪微博@赵望野。



徐飞

2005年至今一直从事企业应用前端架构，对富因特网应用有较深刻的认识，致力于前端的高效开发，研究过Backbone和AngularJS的源码，翻译过讲解AngularJS基本原理的文章，对脏数据检测和基于存取器两种监听方式的差异有深刻认识。

何鹏飞

网名basecss，目前就职于腾讯CDC，任前端工程师。喜欢阅读，喜欢前端技术，崇尚开源。工作之余翻译过Grunt和Lesscss相关文档，同时也是Lesscss中文社区贡献者。

图灵程序设计丛书

ng-book

The Complete Book on AngularJS

AngularJS权威教程

[美] Ari Lerner 著

赵望野 徐飞 何鹏飞 译

人民邮电出版社

北京

图书在版编目（CIP）数据

AngularJS权威教程 / （美）勒纳（Lerner,A.）著 ; 赵望野，徐飞，何鹏飞译. -- 北京 : 人民邮电出版社，2014.8
（图灵程序设计丛书）
ISBN 978-7-115-36647-4

Ⅰ. ①A… Ⅱ. ①勒… ②赵… ③徐… ④何… Ⅲ. ①超文本标记语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2014)第167267号

内容提要

本书是学习 AngularJS 的公认经典，内容全面，讲解通俗，适合各层次的学习者。作者拥有丰富的 AngularJS 开发和教学经验，也是一位全栈工程师。全书 35 章，由浅入深地讲解了 AngularJS 的基本概念和基本功能，包括模块、作用域、控制器、表达式、指令、路由、依赖注入等，重要的是书中对每一个概念的讲解都配合了恰如其分的示例和代码，让读者通过动手实践，切身体会到这些概念的含义和价值。本书后半部分深入到 AngularJS 应用开发，系统地讨论了服务器通信、事件、架构、动画、本地化、安全、缓存、移动应用等主题。

本书适合各个层次的 AngularJS 开发人员学习，无论是出于工作需要，还是好奇心的驱使，只要你想彻底理解 AngularJS，本书都会让你满载而归。

◆ 著　[美] Ari Lerner
译　赵望野　徐　飞　何鹏飞
责任编辑　李松峰
执行编辑　李　静　许林玉
责任印制　焦志炜

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：29.75
字数：760千字　2014年 8 月第 1 版
印数：1－4 000册　2014年 8 月北京第 1 次印刷
著作权合同登记号　图字：01-2014-5140号

定价：99.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第 0021 号

版权声明

在微博上分享这本书

请帮Ari Lerner在新浪微博（http://weibo.com/）上宣传这本书。

推荐本书的微博：

#ngbook#我刚买了《AngularJS权威教程》！我准备构建高级、现代的Webapp！@图灵教育

点击下面这个链接，在新浪微博上搜索其他人对本书的评价：

https://huati.weibo.com/k/ngbook

献　　词

我把这本书献给我的父母，Lisa Lerner和Nelson Lerner，因为没有他们的支持和鼓励就不可能有这本书。

特别感谢

感谢可爱的Q，感谢你一直以来的激励，以及你在编辑方面的过人天赋。感谢我的共同创始人兼朋友Nate Murray。

译　者　序

2012年上半年，我所在的公司正在开发一个二次开发平台，它的目标是从数据库开始，能自由、方便地定制业务数据、规则、流程、服务接口，还有展现层。在对展现层的实现部分，我思考了很久，对其中部分技术细节还是缺乏好的思路，于是把眼光转到开源社区，无意中发现了AngularJS这样一个框架，详细考察之后，我认为它在很大程度上满足了我们的需求，继而投入了不小的精力进行研究。

在这两年里，我差不多遍历了它的源码，了解了很多细节的实现机制，并且与当时研究得较深的几位朋友，比如angularjs.cn的作者严清，资深开发者王宇鹏等进行了交流，获得了很多有益的信息，与此同时，也跟Avalon的作者司徒正美有过一些讨论，对前端MV*有了更深入的认识。后来，团队中的大漠穷秋翻译的《用AngularJS开发下一代Web应用》由电子工业出版社出版。作为国内第一本关于AngularJS的译著，它带动了学习和了解AngularJS框架的浪潮，也因此与朴灵的《深入浅出Node.js》一起，成为前端开发人员拓展思维和技能的两本最受欢迎图书。

到了2014年，我离开工作9年的地方，来到新的工作环境——苏宁云商，本来心里权衡过，很可能不再有使用AngularJS的业务场景了，不曾想到入职之后面对的几个项目都属于云产品，正适合使用这类框架，因此又继续了对它的深入研究。

在此期间，图灵公司的李松峰老师发布了本书招募译者的消息，我心里一动就联系了他。经过沟通之后，我与另外两名译者，豌豆荚的@赵望野和腾讯的@basecss，合作翻译本书，每人负责1/3的内容。第一次正式翻译图书，我很忐忑，翻译过程中也遇到了一些困难。此前我虽翻译过一些技术文章，其中一篇恰好与AngularJS有关，但翻译图书跟翻译文章的差异很大，有很多东西要考虑一致性和连贯性。

本书内容丰富，从零开始向读者讲述AngularJS，首先介绍AngularJS的基本概念，以及在一些场景下的简单应用。接着，本书花很大篇幅讲解AngularJS的周边体系。我们使用这样一个框架，自然需要对前端的架构有一些考虑，包括代码的组织，一些第三方库的选择，甚至还有项目的建立、开发、测试、发布等各环节的综合考虑，这不再是一个简单的编码过程，而是一整套工程化的流程。

另外，我们也可能需要为这样一套前端的技术栈选择相应的后端服务，比如，可以使用Node.js自己建立，或者是利用互联网上已有的一些强大平台（比如Amazon等），在这些平台的帮助下，我们的AngularJS应用将如虎添翼，到达新的高度。

使用一个框架却不去深入了解它的原理，就会一直流于表面，当面对比较复杂的场景时，就找不到优化方案。因此，本书的后面部分也深入剖析了AngularJS的一些原理和拓展主题，比如国际化、移动开发、调试、性能优化等。

无论是零基础的入门级开发者，还是有过一定经验的中高级开发人员，都能从本书中受益。

目前，前端MV*框架百花齐放，AngularJS只是其中较流行的一种。这些框架孰优孰劣，其实并无定论，每个框架都会有它的适用场景，都有它优秀的一面，也没有哪个框架能够通吃所有业务场景，如果因为对一个框架的喜爱，而把它引入到不适合的产品中，一定是有害无益。

因此，我们希望读者在阅读本书时，能够多思考，愿大家在学习本书过程中都能收获满满。这样的话，作为本书译者的我们也将感同身受，与大家一同分享其中的喜悦和满足感。

在本书的出版过程中，除了我们三名译者之外，图灵公司的编辑李静也付出了很大的努力，支付宝的玉伯、51JS版主宝玉、百度的berg提出了不少宝贵意见，对此，一并表示衷心感谢。

徐飞

2014年7月

何鹏飞的个人致谢

感谢我所在的团队，给我提供非常好的环境，让我能学习成长。

感谢我在腾讯的导师@TooBug，让我接触到很多新东西，当然包括这本书所讲述的AngularJS。在翻译的过程中他也为我提供了很多帮助。

感谢图灵公司的出版团队，本书的出版离不开他们的努力和帮助。

最后，还要感谢没有在这里一一列出的帮助过我的每一个人。

赵望野的个人致谢

Web技术日新月异，每天早上翻看各种技术博客，都有一种逆水行舟不进则退的危机感，而这两年来前端MV*框架无疑是Web前端开发领域最热门的话题之一。之前已经读过本书英文版，其中涵盖了AngularJS开发的全部细节，示例代码清晰易懂，因此接下了中文版的翻译工作，并迫不及待地推荐给所有想学习AngularJS的朋友。感谢李松峰老师的帮助，感谢图灵出版团队的辛勤工作，感谢在翻译本书过程中给予帮助的所有朋友。

引　　言

序

似乎每天都有新的JavaScript库或框架发布，对此我多少已经有些麻木了。有能力从众多的库或框架中进行筛选是件好事，但至少在我看来，一个应用程序中如果包含了太多的脚本，对于维护来说却是件坏事。随着应用程序中脚本数量的增加，脚本间会产生依赖关系，所以我一直期待能有那么一到两个脚本，就提供我需要的所有核心功能。

当我第一次听说AngularJS时，它就立刻引起了我的注意，因为它只通过一个独立的框架就可以构建动态、交互密集型的客户端应用。通过进一步的研究，我确信这个第一判断是正确的，于是开始迷上了这个框架。AngularJS提供了一系列健壮的功能，以及将代码隔离成模块的方法，这对提高可复用性、可维护性和可测试性都是非常有益的。它的核心功能包括DOM操作、动画、模板、双向数据绑定、路由、历史管理、Ajax和测试，等等。

基于一个核心框架进行开发虽然很方便，但是学习它却充满挑战。一开始学习AngularJS时，我迷失在各种不同的主题中，并很快变得有些沮丧，甚至开始怀疑它到底是不是我想要的。服务是什么？它和工厂相比有什么区别？作用域服务是怎么同整个系统融合在一起的？指令是什么，我为什么要使用它？将这些零碎的知识点拼在一起形成大局观是我最初要克服的障碍。如果能有一些简明的参考资料，对于降低学习难度大有裨益。

很幸运，你已经有了这样一本优秀的参考资料，就是你手上的这本《AngularJS权威教程》，它将帮助你提升学习效率。本书作者将他掌握的AngularJS知识倾囊相授，并以非常容易理解和学习的方式呈现给大家。如果你想更深入地了解数据绑定、实时模板的工作原理、测试AngularJS应用的流程、服务和工厂的作用以及作用域和控制器如何协同工作等知识，那么这本书就是你所需要的。使用功能强大的AngularJS进行开发是一件非常有趣的事情，本书的示例将帮助你快速掌握这个框架。祝你的AngularJS项目一切顺利！

Dan Wahlin，瓦林咨询公司[①]

致谢

首先，我要感谢一直鼓励我完成这本书的每一个人。那些说写书很容易的人，一定没有亲自写过。

① 个人博客和Twitter页面网址为http://weblogs.asp.net/dwahlin和http://twitter.com/DanWahlin。

我还要亲自感谢Q Kuhns对本书语法方面不厌其烦的修改和支持，感谢Erik Trom耐心地对细节进行修订，以及Nate Murray的清晰思路和乐观精神。

非常感谢Hack Reactor①的全体成员在2013年的暑期课程中给了我一个探索如何在正式场合讲授AngularJS的机会。

同时也要感谢我在30x500的校友们，Sean Iams、Michael Fairchild、Bradly Green、Misko Hevery和整个AirPair团队。

最后，感谢那些对这本书的预览版提供帮助的人。我们从社区获得了非常棒的帮助和支持。特别要感谢以下三位：

- Philip Westwell
- Saurabh Agrawal
- Dougal MacPherson

关于本书

本书包含了能让你成为AngularJS②高手的解决方案。AngularJS是由Google③开发的先进前端框架，借助它你可以快捷高效地开发富交互应用。

本书提供了一系列前沿工具，使你在很短的时间内就可以上手创建令人印象深刻的Web体验。它能帮助你解决棘手的问题，并提供了一些可以立刻投入使用的实用技术。

本书涵盖的主题可以帮助你构建专业的Web应用，并能够非常顺利地执行。这些主题包括：

- 与RESTful风格的Web服务交互；
- 创建可复用的自定义组件；
- 测试；
- 异步编程；
- 创建服务；
- 提供先进的视觉效果；
- 其他更多内容。

本书的目标不仅是让你深刻了解AngularJS的运行原理，而且同时也提供了专业的代码片段，你可以对它们进行修改，从而构建你自己的应用。

借助这些工具和测试，你可以着手使用AngularJS开发自己的动态Web应用了，并且确信你的应用是可扩展的。

本书读者对象

本书写给那些从未使用AngularJS开发过Web应用，并且对如何开始使用这个优秀的框架心存好奇的读者。我们假定读者已经掌握了HTML和CSS，并且熟悉JavaScript（或者其他JavaScript

① http://www.hackreactor.com
② http://angularjs.org
③ http://google.com

框架）的基础知识。

本书组织结构

首先，本书涵盖了入门的基础知识，目的是帮助你很快上手使用AngularJS开发动态Web应用。

接下来会介绍AngularJS的工作原理，以及它与其他流行的JavaScript框架的差异。我们会深入讨论AngularJS应用内部的工作流程。

最后，我们将应用所学的知识开发一个相对复杂的应用程序。

其他资源

我们会引用AngularJS①官方网站的文档。官方文档是非常好的学习资源，我们会经常用到它。

建议你先看一下AngularJS的API文档，通过它，你可以直接获得开发AngularJS应用的推荐方法。同时，这个文档肯定也是最新的。

本书排版约定

本书使用如下排版规范来表示不同类型的信息。

单行代码是这样的：<h1>Hello</h1>

代码块如下所示：

```
var App = angular.module('App', []);

function FirstController($scope) {
    $scope.data = "Hello";
}
```

命令行中的命令如下所示：

```
$ ls -la
```

Chrome（开发过程中使用的主要浏览器）开发者控制台中的命令如下所示：

```
> var obj = {message: "hello"};
```

新术语使用楷体。

重点文字将会加粗。

提示和技巧用如下图标标示：

这个图标表示提示。

提醒和陷阱用警告图标标示：

这个图标表示警告。

① http://angularjs.org

错误信息用如下图标标示：

这个图标表示错误。

重要的补充内容使用如下图标标示：

信息框。

需要讨论的主题用如下图标标示：

这是一个讨论框。

开发环境

为了开发AngularJS应用，首先需要一个顺手的开发环境。在整个学习过程中，我们会将精力主要放在两个环境中：编辑器和浏览器。

本书提到编辑器时指的是你使用的文本编辑器，而浏览器就是你使用的浏览器。强烈建议你下载Google的Chrome浏览器，因为它提供了一个非常强大的开发环境，可以使用开发者工具。

开始之前，我们还需要安装一些库。为了运行测试，我们需要Karma和Node.js。最好也装上git，但不强求。

本书不会介绍如何安装NodeJS。可以访问nodejs.org[①]来获得更多信息[②]。

虽然我们大部分工作都是在浏览器中完成的，但本书的部分内容也会重点介绍如何在服务器端通过构建RESTful风格的API来服务前端。

① http://nodejs.org

② 读者还可以参考图灵公司的《深入浅出Node.js》一书。——编者注

目　　录

第 1 章

初识AngularJS

本章的目标是帮助你熟悉与AngularJS有关的一些术语和技术，以及它们背后相关的工作原理。即使以前从来没有接触过AngularJS，通过将零碎的知识点组合在一起，你也可以构建一个属于自己的AngularJS应用。

1.1 浏览器如何获取网页

我们把互联网想象成一个邮局：当你想给朋友写信时，首先要把内容写在一张信纸上，然后在信封上写上地址，再把信纸装进信封。

当你把信送到邮局，邮件分拣机会根据邮编和地址来判断你的朋友住在哪里。如果他住在一栋有很多房间的公寓大楼里面，邮局会把信件投递到大楼的前台，然后大楼的工作人员会根据房间号再次进行分拣。

互联网的工作原理和上面的过程很类似。不同的是，现实世界中由街道连接起来的楼房和公寓，在互联网世界中被路由器和网线连接起来的计算机所取代。每一台计算机都有一个唯一的地址，让网络可以定位到它。

多个公寓房间共享同一个街道地址，与此类似，多台计算机也可以共享同一个网络或路由器。比如，在使用星巴克提供的免费Wi-Fi时，多台计算机就会共享同一个公网IP地址。尽管如此，你的计算机依然可以通过路由器分配的内网IP地址被单独访问到，路由器就好比公寓大楼的工作人员，而内网IP地址就好比房间号。

> IP是互联网协议（Internet Protocol）的缩写。IP地址是为每个接入到网络中的设备分配的数字标识符。计算机、打印机甚至手机都有自己的IP地址。

> 目前有IPv4和IPv6两种主要的IP地址类型，普遍使用的是IPv4地址，例如192.168.0.199这种形式，而IPv6地址是2001:0db8:0000:0000:0000:ff00:0042:8329这种形式的。

当你打开一个浏览器，并在地址栏输入http://google.com后，浏览器会“询问”网络（更准确地说，是“询问”DNS服务器）google.com对应的IP地址是什么？如果DNS服务器知道你要找的IP地址，就会将其结果返回；如果不知道，它会将请求转发给其他DNS服务器，直到在某一台DNS服务器上找到对应的IP地址记录。在终端输入下列指令，可以观察DNS服务器的响应内容：

```
$ dig google.com
```

> 如果你使用的是Mac操作系统，可以使用Terminal终端程序，它通常储存在/Applications/Utilities目录中。如果使用的是Windows操作系统，打开开始菜单，在运行中输入cmd就可以打开终端了。

DNS服务器返回了你要访问的计算机的IP地址（例如找到了google.com对应的IP地址）后，它就会向这个IP地址对应的计算机请求你要访问的页面。

> 每一个路径对应的网页都由不同的HTML文档组成（也有一些例外）。例如，当浏览器请求http://google.com或http://google.com/images时，得到的HTML文档是不一样的。

现在，计算机已经知道了在哪个IP地址可以访问到http://google.com，它会向Google的服务器请求显示这个页面所需的HTML。

当远程服务器把HTML文档发送回来后，浏览器会对文档进行渲染。渲染就是通过一系列操作，使HTML页面按照设计之初的既定方式显示。

1.2　浏览器是什么

在介绍AngularJS之前，我们需要先了解浏览器在渲染网页的过程中都做了些什么。

目前市场上有很多不同品牌的浏览器，常见的有Chrome、Safari、Firefox和IE。它们的核心功能基本上都是相同的：获取网页，并将它显示给用户。

浏览器获取页面对应的HTML文本，将其解析为一个在浏览器内部使用的结构，对页面的内容进行布局，并在内容显示到屏幕上之前加上样式，所有这些工作都是在浏览器内部进行的。

作为Web开发人员，我们的工作是构造网页的结构和内容，这样浏览器才能将它们转化成对用户来说比较美观的形式。

使用AngularJS，不仅可以构建页面的结构，而且可以构建用户和Web应用之间的交互。

1.3　AngularJS 是什么

AngularJS的官方文档是这样介绍它的。

> 完全使用JavaScript编写的客户端技术。同其他历史悠久的Web技术（HTML、CSS和JavaScript）配合使用，使Web应用开发比以往更简单、更快捷。

AngularJS主要用于构建单页面Web应用。它通过增加开发人员和常见Web应用开发任务之间的抽象级别，使构建交互式的现代Web应用变得更加简单。

> AngularJS的开发团队将其描述为一种构建动态Web应用的结构化框架。

AngularJS使开发Web应用变得非常简单，同时也降低了构建复杂应用的难度。它提供了开发者在现代Web应用中经常要用到的一系列高级功能，例如：

- 解耦应用逻辑、数据模型和视图；
- Ajax服务；

- 依赖注入；
- 浏览历史（使书签和前进、后退按钮能够像在普通Web应用中一样工作）；
- 测试；
- 更多功能。

1.3.1 AngularJS有什么不同

在其他JavaScript框架中，我们被迫从自定义的JavaScript对象中进行扩展，并从外到内操作DOM。以jQuery[①]为例，为了在DOM中插入一个按钮元素，我们必须知道要把元素放到何处，并在合适的位置插入它：

```
var btn = $("<button>Hi</button>");
btn.on('click', function(evt) { console.log("Clicked button"); });
$("#checkoutHolder").append(btn);
```

尽管这个过程并不复杂，但是它要求开发者对整个DOM结构都有所了解，并强迫我们在JavaScript代码中加入复杂的控制逻辑，用以操作外部DOM。

而AngularJS则通过原生的Model-View-Controller（MVC，模型–视图–控制器）功能增强了HTML。结果表明，这个选择可以快捷和愉悦地构建出令人印象深刻并且极富表现力的客户端应用。

利用它，开发者可将页面的一部分封装为一个应用，并且不强迫整个页面都使用AngularJS进行开发。这个特质在某些情况下非常有用，比如你的工作流程中已经包含了另外一个框架，或者你只希望页面中的某一部分是动态的，而剩下的部分是静态的或者是由其他JavaScript框架来控制的。

此外，AngularJS团队非常重视框架文件压缩后的大小，这样使用它就不会付出太多的额外代价（写作本书时，文件压缩后的体积在90 KB左右）。这一特性使得AngularJS非常适合用于开发功能原型。

1.3.2 许可

AngularJS的源码托管在GitHub[②]上，可以免费获取。它基于MIT许可发布，这意味着你可以为AngularJS贡献代码，使其变得更加优秀。

为了促进大家为AngularJS贡献代码，开发团队把开发流程变得相对开放。任何重大变化都需要在AngularJS的邮件列表上[③]进行讨论，所有人都可以加入讨论，这样一来大家就可以对潜在的变动进行改进，并且防止重复劳动。

关于贡献代码的更多内容可以在AngularJS的官网中查看“贡献代码”部分[④]。

---

① http://jquery.com/

② http://github.com

③ https://groups.google.com/forum/?hl=en#!forum/angular

④ http://docs.angularjs.org/misc/contribute

第 2 章

数据绑定和第一个 AngularJS Web应用

Hello World

写一个Hello World应用是开始学习AngularJS的最基本途径，让我们从一段简单得不能再简单的HTML开始吧。

随着学习的深入，我们会逐渐深入到AngularJS的内部原理中。现在，让我们先来写一个Hello World应用。

```
<!DOCTYPE html>
<html ng-app>
<head>
  <title>Simple app</title>
  <script
    src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.13/angular.js">
    </script>
</head>
<body>
  <input ng-model="name" type="text" placeholder="Your name">
  <h1>Hello {{ name }}</h1>
</body>
</html>
```

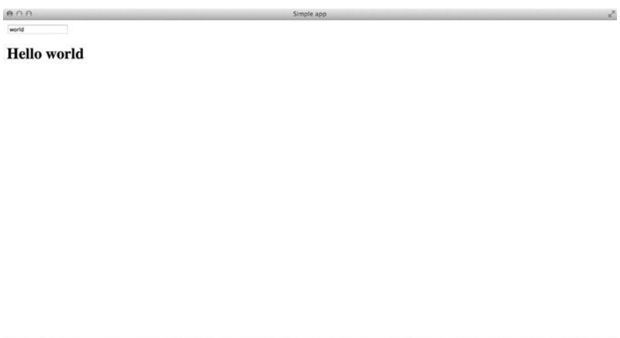


图2-1　Hello World

虽然这个例子不怎么有趣，但它展示了AngularJS最基本也最令人印象深刻的功能之一：数据绑定。

请注意，在本章的示例代码中为了方便展示第一个核心概念，我们并没有用最佳方式编写控制器。这也是本书唯一一处我们不鼓励将示例代码应用在实际生产中，而只作为学习范例的地方。

2.1 AngularJS 中的数据绑定

在Rails等传统Web框架中，控制器将多个模型中的数据和模板组合在一起形成视图，并将其提供给用户，这个组合过程会产生一个单向视图。如果没有创建任何自定义的JavaScript组件，视图只会体现它渲染时模型暴露出的数据。在写这篇文章时，已经出现了好几个可以在视图和模型之间自动进行数据绑定的框架。

AngularJS则采用了完全不同的解决方案。它创建实时模板来代替视图，而不是将数据合并进模板之后更新DOM。任何一个独立视图组件中的值都是动态替换的。这个功能可以说是AngularJS中最重要的功能之一，也是让我们只用10行代码，并且在没有任何JavaScirpt的情况下就可以写出Hello World的关键。

要实现这个功能，只要在HTML页面中引用`angular.js`，并在某个DOM元素上明确设置ng-app属性即可。ng-app属性声明所有被其包含的内容都属于这个AngularJS应用，这也是我们可以在Web应用中嵌套AngularJS应用的原因。只有被具有ng-app属性的DOM元素包含的元素才会受AngularJS影响。

视图中的插值会在计算一个或多个变量时被动态替换，替换结果是字符串中的插值被变量的值替代。

例如，如果有一个叫做name的变量，它的值是“Ari”，那么视图中的`"Hello {{ name }}"`字符串会被替换成“Hello Ari”。

自动数据绑定使我们可以将视图理解为模型状态的映射。当客户端的数据模型发生变化时，视图就能反映出这些变化，并且不需要写任何自定义的代码，它就可以工作。

在MVC（Model View Controller，模型–视图–控制器）的世界里，控制器可以不必担心会牵扯到渲染视图的工作。这样我们就不必再担心如何分离视图和控制器逻辑，并且也可以使测试变得既简单又令人愉悦。

MVC是一种软件架构设计模式，它将表现从用户交互中分离出来。通常来讲，模型中包含应用的数据和与数据进行交互的方法，视图将数据呈献给用户，而控制器则是二者之间的桥梁。

这种表现分离[1]能将应用中的对象很好地隔离开来，因此视图不需要知道如何保存对象，只要知道如何显示它即可。这也意味着数据模型不需要同视图进行交互，只需要包含数据和操作视图的方法。控制器用来存放将二者绑定在一起的业务逻辑。

① http://martinfowler.com/eaaDev/uiArchs.html

AngularJS[1]会记录数据模型所包含的数据在任何特定时间点的值（在Hello World例子中就是name的值），而不是原始值。

当AngularJS认为某个值可能发生变化时，它会运行自己的事件循环来检查这个值是否变“脏”。如果该值从上次事件循环运行之后发生了变化，则该值被认为是“脏”值。这也是Angular可以跟踪和响应应用变化的方式。

这个事件循环会调用$digest()循环，第23章将会详细介绍。

这个过程被称作脏检查（dirty checking）。脏检查是检查数据模型变化的有效手段。当有潜在的变化存在时，AngularJS会在事件循环时执行脏检查（第24章会深入讨论）来保证数据的一致性。

如果使用KnockoutJS这种通过在数据模型上绑定事件监听器来监听数据变化的框架，这个过程会变得更复杂且低效[2]。处理事件合并、依赖跟踪和大量的事件触发（event firing）是非常复杂的，而且会在性能方面导致额外的问题。

尽管存在更高效的方式，但脏检查可以运行在所有浏览器中并且是可预测的。此外，很多在速度和效率方面有要求的软件都会使用脏检查的方案[3]。

借助AngularJS，不需要构建复杂和新的JavaScript功能，就可以在视图中实现类自动同步的机制。

为了表示内部和内置的库函数，Angular使用$预定义对象。尽管这类似于全局的jQuery对象$，但它们是完全无关的。只要遇到$符号，你都可以只把它看作一个Angular对象。

2.2　简单的数据绑定

审阅一下上面写的代码，我们使用ng-model指令将内部数据模型对象（$scope）中的name属性绑定到了文本输入字段上。

这意味着无论在文本输入字段中输入了什么，都会同步到数据模型中。

数据模型对象（model object）是指$scope对象。$scope对象是一个简单的JavaScript对象，其中的属性可以被视图访问，也可以同控制器进行交互。如果不理解这个概念也没有关系，后面的例子将会对这个概念进行详细说明。
双向数据绑定（bi-directional）意味着如果视图改变了某个值，数据模型会通过脏检查观察到这个变化，而如果数据模型改变了某个值，视图也会依据变化重新渲染。

在输入字段上使用ng-model指令来实现数据绑定，如下所示：

---

① http://angularjs.org
② 这有些言过其实，低效是真，复杂未必。——译者注
③ 比如在游戏开发中就大量使用脏检查技术。——译者注

```
<input ng-model="person.name" type="text" placeholder="Yourname">
<h1>Hello{{ person.name }}</h1>
```

这样绑定就设置好了（没错，就是这么简单）。我们可以观察一下视图是如何更新数据模型的。当输入字段中的值发生变化时，person.name会被更新，而视图将反映出这个更新。

我们仅通过视图就实现了一个双向数据绑定。为了从其他角度（后端到前端）解释双向数据绑定，后面会深入介绍控制器。

正如ng-app声明所有被它包含的元素都属于AngularJS应用一样，DOM元素上的ng-controller声明所有被它包含的元素都属于某个控制器。

为了解释这个概念，我们将前面的例子修改成如下的样子：

```
<div ng-controller='MyController'>
  <input ng-model="person.name" type="text" placeholder="Your name">
  <h1>Hello {{ person.name }}</h1>
</div>
```

在这个例子中，我们会创建一个每秒钟走一步的时钟（时钟通常都是这样的），并更新clock变量上的数据：

```
function MyController($scope, $timeout) {
  var updateClock = function() {
    $scope.clock = new Date();
    $timeout(function() {
      updateClock();
    }, 1000);
  };
  updateClock();
};
```

在这个例子中，MyController函数接受两个参数，即该DOM元素的$scope和$timeout。第13章将介绍如何使用不同的变量定义函数。

在这个例子中，当定时器触发时会调用updateClock()函数，将$scope.clock的值设置为当前时间。

- $timeout对象

可以在视图中将clock变量用{{ }}包起来，以显示$scope中的clock的值：

```
<div ng-controller="MyController">
  <h5>{{ clock }}</h5>
</div>
```

下面是完整的示例代码：

```
<!doctype html>
<html ng-app>
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.13/angular.js"></script>
  </head>
  <body>
    <div ng-controller="MyController">
      <h1>Hello {{ clock }}!</h1>
```

```
    </div>
    <script type="text/javascript">
    function MyController($scope, $timeout) {
        var updateClock = function() {
          $scope.clock = new Date();
          $timeout(function() {
            updateClock();
          }, 1000);
        };
        updateClock();
      };
    </script>
  </body>
</html>
```

在线示例：http://jsbin.com/uHiVOZo/1/edit?html,output。

尽管我们可以将所有代码都写在一个文件中，但由于需要将不同的组件分开开发，将代码写在一个文件中会使协同工作变得非常困难。通常情况下，更好的选择是将JavaScript放在单独的文件中，而不是index.html中。

上面的代码可以修改成：

```
<!doctype html>
<html ng-app>
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.13/angular.js"></script>
  </head>
  <body>
    <div ng-controller="MyController">
      <h1>Hello {{ clock }}!</h1>
    </div>
    <script type="text/javascript" src="js/app.js"></script>
   </body>
</html>
```

将前面例子中的JavaScript代码放在js/app.js文件中，而不是将它直接写在HTML中。

```
// 在app.js中
function MyController($scope, $timeout) {
  var updateClock = function() {
    $scope.clock = new Date();
    $timeout(function() {
      updateClock();
    }, 1000);
  };
  updateClock();
};
```

2.3　数据绑定的最佳实践

由于JavaScript自身的特点，以及它在传递值和引用时的不同处理方式，通常认为，在视图中通过对象的属性而非对象本身来进行引用绑定，是Angular中的最佳实践。

如果把这个最佳实践应用到上面时钟的例子中，需要把视图中的代码改写成下面这样：

```
<!doctype html>
<html ng-app>
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.13/angular.js"></script>
  </head>
  <body>
    <div ng-controller="MyController">
      <h1>Hello {{ clock.now }}!</h1>
    </div>
    <script type="text/javascript" src="js/app.js"></script>
   </body>
</html>
```

在这个例子中，相比每秒钟都更新$scope.clock，更新clock.now的值会是更好的选择。有了这个优化后，我们将反映数据变化的逻辑做如下修改：

```
// 在app.js中
function MyController($scope) {
    $scope.clock = {
        now: new Date()
    };
    var updateClock = function() {
        $scope.clock.now = new Date()
    };
    setInterval(function() {
        $scope.$apply(updateClock);
    }, 1000);
    updateClock();
};
```

将所有绑定都通过这样的形式放在视图中，是个非常好的主意。

第 3 章 模　　块

在JavaScript中，将函数代码全部都定义在全局命名空间中绝对不是什么好主意，这样做会导致冲突从而使调试变得非常困难，浪费宝贵的开发时间。

上一章介绍数据绑定时，我们把控制器的代码写到了一个在全局命名空间中定义的函数里：

```
function MyController($scope) {
  var updateClock = function() {
    $scope.clock = new Date();
  };
  setInterval(function() {
    $scope.$apply(updateClock);
  }, 1000);
  updateClock();
};
```

本章将讨论如何写出高效、能用在生产环境中的控制器代码，并把它封装在一个我们称之为模块（module）的单元内。

在AngularJS中，模块是定义应用的*最主要*方式。模块包含了主要的应用代码。一个应用可以包含多个模块，每一个模块都包含了定义具体功能的代码。

使用模块能给我们带来许多好处，比如：

- 保持全局命名空间的清洁；
- 编写测试代码更容易，并能保持其清洁，以便更容易找到互相隔离的功能；
- 易于在不同应用间复用代码；
- 使应用能够以任意顺序加载代码的各个部分。

AngularJS允许我们使用`angular.module()`方法来声明模块，这个方法能够接受两个参数，第一个是模块的名称，第二个是依赖列表，也就是可以被注入到模块中的对象列表。

```
angular.module('myApp', []);
```

这个方法相当于AngularJS模块的setter方法，是用来定义模块的。

调用这个方法时如果只传递一个参数，就可以用它来引用模块。例如，可以通过以下代码来引用myApp模块：

```
// 这个方法用于获取应用
angular.module('myApp')
```

> 这个方法相当于AngularJS模块的getter方法，用来获取对模块的引用。

接下来，就可以在`angular.module('myApp')`返回的对象上创建我们的应用了。

开发大型应用时，我们会创建多个模块来承载业务逻辑。将复杂的功能分割成不同的模块，有助于单独为它们编写测试，相关信息参见第21章。

3.1　参数

下面是`angular.module()`的参数列表。

3.1.1　name（字符串）

`name`是模块的名称，字符串变量。

3.1.2　requires（字符串数组）

`requires`包含了一个字符串变量组成的列表，每个元素都是一个模块名称，本模块依赖于这些模块，依赖需要在本模块加载之前由注入器进行预加载。

第 4 章 作用域

作用域（scope）[①]是构成AngularJS应用的核心基础，在整个框架中都被广泛使用，因此了解它如何工作是非常重要的。

应用的作用域是和应用的数据模型相关联的，同时作用域也是表达式执行的上下文。`$scope`对象是定义应用业务逻辑、控制器方法和视图属性的地方。

作用域是视图和控制器之间的胶水。在应用将视图渲染并呈献给用户之前，视图中的模板会和作用域进行连接，然后应用会对DOM进行设置以便将属性变化通知给AngularJS。这个功能让XHR请求等promise对象的实现变得非常容易。查看第17章获取更多关于promise对象的内容。

作用域是应用状态的基础。基于动态绑定，我们可以依赖视图在修改数据时立刻更新`$scope`，也可以依赖`$scope`在其发生变化时立刻重新渲染视图。

AngularJS将`$scope`设计成和DOM类似的结构，因此`$scope`可以进行嵌套，也就是说我们可以引用父级`$scope`中的属性。

> 如果你了解JavaScript，对这个分层的概念应该并不陌生。在JavaScript中，当创建一个新的执行上下文时，实际上是用函数创建了一个新的本地上下文。AngularJS中`$scope`的概念与其类似，当为子DOM元素创建新的作用域时，实际上是为子DOM元素创建了一个新的执行上下文。

作用域提供了监视数据模型变化的能力。它允许开发者使用其中的`apply`机制，将数据模型的变化在整个应用范围内进行通知。我们在作用域的上下文中定义和执行表达式，同时它也是将事件通知给另一个控制器和应用其他部分的中介。

将应用的业务逻辑都放在控制器中，而将相关的数据都放在控制器的作用域中，这是非常完美的架构。

4.1 视图和$scope的世界

AngularJS启动并生成视图时，会将根`ng-app`元素同`$rootScope`进行绑定。`$rootScope`是所有`$scope`对象的最上层。

---

① 如非特别强调，本书中的作用域均指AngularJS中的作用域对象，而不是JavaScript作用域。——译者注

$rootScope是AngularJS中最接近全局作用域的对象。在$rootScope上附加太多业务逻并不是好主意，这与污染JavaScript的全局作用域是一样的。

$scope对象就是一个普通的JavaScript对象，我们可以在其上随意修改或添加属性。

$scope对象在AngularJS中充当数据模型，但与传统的数据模型不一样，$scope并不负责处理和操作数据，它只是视图和HTML之间的桥梁，它是视图和控制器之间的胶水。

$scope的所有属性，都可以自动被视图访问到。假设我们有如下的HTML：


```
<div ng-app="myApp">
  <h1>Hello {{ name }}</h1>
</div>
```

我们希望{{ name }}变量是本地$scope的一个属性，效果如图4-1所示。

```
angular.module('myApp', [])
  .run(function($rootScope) {
    $rootScope.name = "World";
});
```



图4-1 简单的$rootScope绑定

4.2 就是 HTML 而已

我们的应用负责渲染HTML并将它交给浏览器来显示。这个HTML中包含了各种标准的HTML元素，包括AngularJS特有的以及非AngularJS特有的元素。AngularJS不会对不包含AngularJS特殊声明的元素进行任何处理。

```
<h2>Hello world</h2>
<h3>Hello {{ name }}</h3>
```

上面这个例子中，AngularJS不会处理<h2>元素，但是会在作用域发生变化时更新<h3>元素。

我们可以在AngularJS应用的模板中使用多种标记，包括下面这些。

- 指令：将DOM元素增强为可复用的DOM组件的属性或元素。
- 值绑定：模板语法{{ }}可以将表达式绑定到视图上。
- 过滤器：可以在视图中使用的函数，用来进行格式化。
- 表单控件：用来检验用户输入的控件。

4.3　作用域能做什么

作用域有以下的基本功能：

- 提供观察者以监视数据模型的变化；
- 可以将数据模型的变化通知给整个应用，甚至是系统外的组件；
- 可以进行嵌套，隔离业务功能和数据；
- 给表达式提供运算时所需的执行环境。

开发AngularJS应用的大部分工作内容，就是构建作用域及其相关的功能。

> 作用域包含了渲染视图时所需的功能和数据，它是所有视图的唯一源头。可以将作用域理解成视图模型（view model）。

前面的例子中，我们在$rootScope中设置了一个name变量并在视图中引用了它：

```
angular.module('myApp', [])
  .run(function($rootScope) {
    $rootScope.name = "World";
});
```

在视图中可以引用这个name属性并将它展示给用户：

```
<div ng-app="myApp">
  <h1>Hello {{ name }}</h1>
</div>
```

我们可以不将变量设置在$rootScope上，而是用控制器显式创建一个隔离的子$scope对象，把它设置到这个子对象上。使用ng-controller指令可以将一个控制器对象附加到DOM元素上，如下所示：

```
<div ng-app="myApp">
  <div ng-controller="MyController">
    <h1>Hello {{ name }}</h1>
  </div>
</div>
```

我们可以创建一个控制器来管理与其相关的变量，而不用将name变量直接放在$rootScope上：

```
angular.module("myApp", [])
.controller('MyController',
function($scope) {
  $scope.name = "Ari";
});
```

ng-controller指令为这个DOM元素创建了一个新的$scope对象，并将它嵌套在$rootScope中。

4.4　$scope 的生命周期

当Angular关心的事件发生在浏览器中时，比如用户在通过ng-model属性监控的输入字段中输入，或者带有ng-click属性的按钮被点击时，Angular的事件循环都会启动。这个事件将在Angular执行上下文中处理。

更多关于Angular执行上下文的信息请参考第23章。

每当事件被处理时，$scope就会对定义的表达式求值。此时事件循环会启动，并且Angular应用会监控应用程序内的所有对象，脏值检测循环也会运行。

第6章将深入讨论表达式。作用域的表达式就是赋值给作用域对象的变量。当我们给上面提到的作用域中的name变量赋值，比如$scope.name="Ari"，实际上是设置了一个表达式，即使这个值只是一个简单的字符串。

$scope对象的生命周期处理有四个不同阶段。

4.4.1 创建

在创建控制器或指令时，AngularJS会用$injector创建一个新的作用域，并在这个新建的控制器或指令运行时将作用域传递进去。

4.4.2 链接

当Angular开始运行时，所有的$scope对象都会附加或者链接到视图中。所有创建$scope对象的函数也会将自身附加到视图中。这些作用域将会注册当Angular应用上下文中发生变化时需要运行的函数。

这些函数被称为$watch函数，Angular通过这些函数获知何时启动事件循环。

4.4.3 更新

当事件循环运行时，它通常执行在顶层$scope对象上（被称作$rootScope），每个子作用域都执行自己的脏值检测。每个监控函数都会检查变化。如果检测到任意变化，$scope对象就会触发指定的回调函数。

4.4.4 销毁

当一个$scope在视图中不再需要时，这个作用域将会清理和销毁自己。

尽管永远不会需要清理作用域（因为Angular会为你处理），但是知道是谁创建了这个作用域还是有用的，因为你可以使用这个$scope上叫做$destory()的方法来清理这个作用域。

4.5 指令和作用域

指令在AngularJS中被广泛使用，指令通常不会创建自己的$scope，但也有例外。比如ng-controller和ng-repeat指令会创建自己的子作用域并将它们附加到DOM元素上。

在介绍更多内容之前，我们先来看看控制器是什么，以及如何在应用中使用它们。

第 5 章

控制器

控制器在AngularJS中的作用是增强视图。在Hello World的例子中，我们并没有使用普通的控制器，而是使用了一个隐式控制器。

AngularJS中的控制器是一个*函数*，用来向视图的作用域中添加额外的功能。我们用它来给作用域对象设置初始状态，并添加自定义行为。

当我们在页面上创建一个新的控制器时，AngularJS会生成并传递一个新的$scope给这个控制器。可以在这个控制器里初始化$scope。由于AngularJS会负责处理控制器的实例化过程，我们只需编写构造函数即可。

下面的例子展示了控制器初始化：

```
function FirstController($scope) {
  $scope.message = "hello";
}
```

将控制器命名为[Name]Controller而不是[Name]Ctrl是一个最佳实践。

正如我们看到的那样，AngularJS会在创建作用域时调用控制器方法。

细心的读者会发现，我们是在全局作用域中创建的这个函数。这样做并不合适，因为会污染全局命名空间。更合理的方式是创建一个模块，然后在模块中创建控制器，如下所示：

```
var app = angular.module('app', []);
app.controller('FirstController', function($scope) {
  $scope.message = "hello";
});
```

只需创建控制器作用域中的函数，就能创建可以在视图中使用的自定义操作。很幸运，AngularJS允许我们在视图中像调用普通数据一样调用$scope上的函数。

用内置指令ng-click可以将按钮、链接等其他任何DOM元素同点击事件进行绑定。ng-click指令将浏览器中的mouseup事件，同设置在DOM元素上的事件处理程序绑定在一起（例如，当浏览器在某个DOM元素上触发了点击事件，函数就会被调用）。和前面的例子类似，绑定看起来是这样的：

```
<div ng-controller="FirstController">
  <h4>The simplest adding machine ever</h4>
  <button ng-click="add(1)" class="button">Add</button>
  <a ng-click="subtract(1)" class="button alert">Subtract</a>
  <h4>Current count: {{ counter }}</h4>
</div>
```

按钮和链接都被绑定在了内部$scope的一个操作上，当点击任何一个元素时AngularJS都会调用相应的方法。注意，当设置调用哪个函数时，会同时用括号传递一个参数（add(1)）。

下面给FirstController添加一个操作：

```
app.controller('FirstController', function($scope) {
  $scope.counter = 0;
  $scope.add = function(amount) { $scope.counter += amount; };
  $scope.subtract = function(amount) { $scope.counter -= amount;  };
});
```

用这种设置方式我们可以在视图中调用add()或subtract()方法，这两个方法可以定义在FirstController的作用域中，或其父级的$scope中。

控制器可以将与一个独立视图相关的业务逻辑封装在一个独立的容器中。尽可能地精简控制器是很好的做法。作为AngularJS开发者，使用依赖注入来访问服务可以实现这个目的。

AngularJS同其他JavaScript框架最主要的一个区别就是，控制器并不适合用来执行DOM操作、格式化或数据操作，以及除存储数据模型之外的状态维护操作。它只是视图和$scope之间的桥梁。

AngularJS允许在$scope上设置包括对象在内的任何类型的数据，并且在视图中还可以展示对象的属性。

例如，我们在MyController上创建一个person对象，这个对象只有name这一个属性：

```
app.controller('MyController', function($scope) {
  $scope.person = {
    name: 'Ari Lerner'
  };
});
```

在拥有ng-controller='MyController'这个属性的元素内部的任何子元素中，都可以访问person对象，因为它是定义在$scope上的。

例如，可以方便地在视图中引用person或person.name，效果如图5-1所示。

```
<div ng-app="myApp">
  <div ng-controller="MyController">
    <h1>{{ person }}</h1>
    and their name:
    <h2>{{ person.name }}</h2>
  </div>
</div>
```

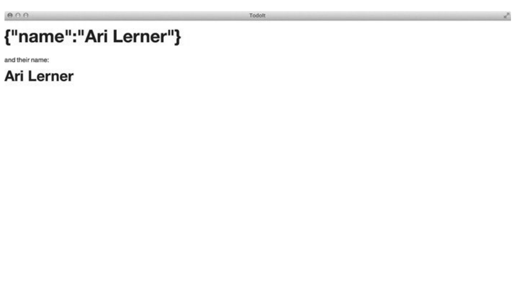


图5-1　控制器对象

正如看到的这样，$scope对象用来从数据模型向视图传递信息。同时，它也可以用来设置事件监听器，同应用的其他部分进行交互，以及创建与应用相关的特定业务逻辑。

AngularJS通过作用域将视图、控制器和指令（本书后面会介绍）隔离开来，这样就很容易为功能的具体部分编写测试。

5.1　控制器嵌套（作用域包含作用域）

AngularJS应用的任何一个部分，无论它渲染在哪个上下文中，都有父级作用域存在。对于ng-app所处的层级来讲，它的父级作用域就是$rootScope。

有一个例外：在指令内部创建的作用域被称作孤立作用域。

除了孤立作用域外，所有的作用域都通过原型继承而来，也就是说它们都可以访问父级作用域。如果熟悉面向对象编程，对这个机制应该不会陌生。

默认情况下，AngularJS在当前作用域中无法找到某个属性时，便会在父级作用域中进行查找。如果AngularJS找不到对应的属性，会顺着父级作用域一直向上寻找，直到抵达$rootScope为止。如果在$rootScope中也找不到，程序会继续运行，但视图无法更新。

通过例子来看一下这个行为。创建一个ParentController，其中包含一个user对象，再创建一个ChildController来引用这个对象：

```
app.controller('ParentController', function($scope) {
  $scope.person = {greeted: false};
});

app.controller('ChildController', function($scope) {
  $scope.sayHello = function() {
    $scope.person.name = 'Ari Lerner';
  };
});
```

如果我们将ChildController置于ParentController内部，那ChildController的$scope对象的父级作用域就是ParentController的$scope对象。根据原型继承的机制，我们可以在子作用域中访问ParentController的$scope对象。

例如，我们可以在ChildController的DOM元素中访问定义在ParentController中的person对象，如图5-2所示。

```
<div ng-controller="ParentController">
  <div ng-controller="ChildController">
    <a ng-click="sayHello()">Say hello</a>
  </div>
  {{ person }}
</div>
```

控制器的这种嵌套结构和DOM的嵌套结构很相似。

我们看到，点击按钮时，可以在ChildController中访问ParentController中$scope.person的

值，就好像person对象定义在ChildController的$scope中一样。

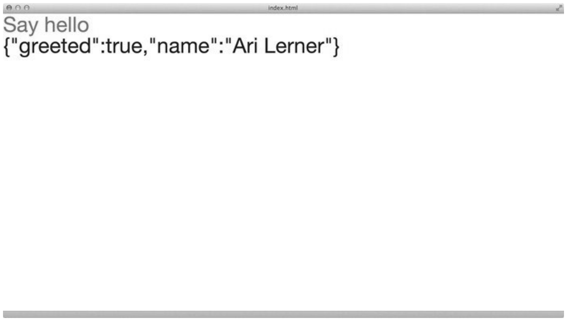


图5-2 控制器嵌套

控制器应该尽可能保持短小精悍，而在控制器中进行DOM操作和数据操作则是一个不好的实践。

例如，下面这个例子中的控制器包含了过于臃肿的逻辑用于控制视图，并且还操作了DOM。

臃肿的控制器：

```
angular.module('myApp', [])
.controller('MyController', function($scope) {
  $scope.shouldShowLogin = true;
  $scope.showLogin = function () {
     $scope.shouldShowLogin = !$scope.shouldShowLogin;
  };
  $scope.clickButton = function() {
     $('#btn span').html('Clicked');
  };
  $scope.onLogin = function(user) {
     $http({
     method: 'POST',
     url: '/login',
     data: {
       user: user
     }
   }).success(function(data) {
      // user
   });
  };
});
```

设计良好的应用会将复杂的逻辑放到指令和服务中。通过使用指令和服务，我们可以将控制器重构成一个轻量且更易维护的形式：

简洁的控制器：

```
angular.module('myApp', [])
.controller('MyController', function($scope,UserSrv) {
  // 内容可以被指令控制
  $scope.onLogin = function(user) {
    UserSrv.runLogin(user);
  };
});
```

第 6 章 表达式

表达式在AngularJS应用中被广泛使用，因此深入理解AngularJS如何使用并运算表达式是非常重要的。

前面已经见过使用表达式的示例。用`{{ }}`符号将一个变量绑定到`$scope`上的写法本质上就是一个表达式：`{{ expression }}`。当用`$watch`进行监听时，AngularJS会对表达式或函数进行运算。

表达式和`eval(javascript)`非常相似，但是由于表达式由AngularJS来处理，它们有以下显著不同的特性：

- 所有的表达式都在其所属的作用域内部执行，并有访问本地`$scope`的权限；
- 如果表达式发生了`TypeError`和`ReferenceError`并不会抛出异常；
- 不允许使用任何流程控制功能（条件控制，例如`if/eles`）；
- 可以接受过滤器和过滤器链。

对表达式进行的任何操作，都会在其所属的作用域内部执行，因此可以在表达式内部调用那些限制在此作用域内的变量，并进行循环、函数调用、将变量应用到数学表达式中等操作。

6.1 解析 AngularJS 表达式

尽管AngularJS会在运行`$digest`循环的过程中自动解析表达式，但有时手动解析表达式也是非常有用的。

AngularJS通过`$parse`这个内部服务来进行表达式的运算，这个服务能够访问当前所处的作用域。这个过程允许我们访问定义在`$scope`上的原始JavaScript数据和函数。

将`$parse`服务注入到控制器中，然后调用它就可以实现手动解析表达式。举例来说，如果页面上有一个输入框绑定到了`expr`变量上，如下所示：

```
<div ng-controller="MyController">
  <input ng-model="expr"
         type="text"
         placeholder="Enter an expression" />
  <h2>{{ parseValue }}</h2>
</div>
```

我们可以在`MyController`中给`expr`这个表达式设置一个`$watch`并解析它：

```
angular.module("myApp", [])
.controller('MyController',
function($scope,$parse) {
  $scope.$watch('expr', function(newVal, oldVal, scope) {
    if (newVal !== oldVal) {
      // 用该表达式设置parseFun
      var parseFun = $parse(newVal);
      // 获取经过解析后表达式的值
      $scope.parsedValue = parseFun(scope);
    }
  });
});
```

在线示例：http://jsbin.com/UWuLALOf/1/edit?html,js,output。

6.2　插值字符串

在AngularJS中，我们的确有手动运行模板编译的能力。例如，插值允许基于作用域上的某个条件实时更新文本字符串。

要在字符串模板中做插值操作，需要在你的对象中注入`$interpolate`服务。在下面的例子中，我们将会将它注入到一个控制器中：

```
angular.module('myApp', [])
  .controller('MyController',
    function($scope, $interpolate) {
        // 我们同时拥有访问$scope和$interpolate服务的权限
});
```

`$interpolate`服务是一个可以接受三个参数的函数，其中第一个参数是必需的。

- ❑ `text`（字符串）：一个包含字符插值标记的字符串。
- ❑ `mustHaveExpression`（布尔型）：如果将这个参数设为`true`，当传入的字符串中不含有表达式时会返回`null`。
- ❑ `trustedContext`（字符串）：AngularJS会对已经进行过字符插值操作的字符串通过`$sec.getTrusted()`方法进行严格的上下文转义。

查看29.4节以获得关于最后一个参数的更多细节内容。

`$interpolate`服务返回一个函数，用来在特定的上下文中运算表达式。

设置好这些参数后，就可以在控制器中进行字符插值的操作了。例如，假设我们希望可以在电子邮件的正文中进行实时编辑，当文本发生变化时进行字符插值操作并将结果展示出来。

```
<div ng-controller="MyController">
  <input ng-model="to"
        type="email"
        placeholder="Recipient" />
  <textarea ng-model="emailBody"></textarea>
  <pre>{{ previewText }}</pre>
</div>
```

由于控制器内部设置了一个需要每次变化都重新进行字符插值的自定义输入字段，因此需要设置一个$watch来监听数据的变化。第23章将深入讨论$watch。为了保证示例的完整性，在这里我们为$watch引入完整的代码。

> 简而言之，$watch函数会监视$scope上的某个属性。只要属性发生变化就会调用对应的函数。可以使用$watch函数在$scope上某个属性发生变化时直接运行一个自定义函数。

在控制器中，我们设置了$watch来监视邮件正文的变化，并将emailBody属性的值进行字符插值后的结果赋值给previewText属性。

```
angular.module('myApp', [])
  .controller('MyController', function($scope, $interpolate) {
    // 设置监听
    $scope.$watch('emailBody', function(body) {
      if (body) {
        var template = $interpolate(body);
        $scope.previewText =
          template({to: $scope.to});
      }
    };
});
```

> 在线实例：http://jsbin.com/oDeFuCAW/1/edit?html,js,output。

现在，在{{ previewText }}内部的文本中可以将{{ to }}当做一个变量来使用，并对文本的变化进行实时更新。

> 如果需要在文本中使用不同于{{ }}的符号来标识表达式的开始和结束，可以在$inter polateProvider中配置。

用startSymbol()方法可以修改标识开始的符号。这个方法接受一个参数。

- value（字符型）：开始符号的值。

用endSymbol()方法可以修改标识结束的符号。这个方法也接受一个参数。

- value（字符型）：结束符号的值。

如果要修改这两个符号的设置，需要在创建新模块时将$interpolateProvider注入进去。

下面我们来创建一个服务，第14章会对服务进行深入讨论。

```
angular.module('emailParser', [])
  .config(['$interpolateProvider', function($interpolateProvider) {
      $interpolateProvider.startSymbol('__');
      $interpolateProvider.endSymbol('__');
}])
.factory('EmailParser', ['$interpolate', function($interpolate) {
    // 处理解析的服务
    return {
      parse: function(text, context) {
        var template = $interpolate(text);
        return template(context);
```

```
        }
      };
    }]);
```

现在，我们已经创建了一个模块，可以将它注入到应用中，并在邮件正文的文本中运行这个邮件解析器：

```
angular.module('myApp', ['emailParser'])
  .controller('MyController', ['$scope', 'EmailParser',
      function($scope, EmailParser) {
        // 设置监听
        $scope.$watch('emailBody', function(body) {
          if (body) {
            $scope.previewText = EmailParser.parse(body, {
                to: $scope.to
            });
          }
      });
}]);
```


现在用自定义的 __ 符号取代默认语法中的 {{ }} 符号来请求插值文本。

由于我们将表达式开始和结束的符号都设置成了__，因此需要将HTML修改成用这个符号取代{{ }}的版本，效果如图6-1所示。

```
<div id="emailEditor">
    <input ng-model="to"
        type="email"
        placeholder="Recipient" />
    <textarea ng-model="emailBody"></textarea>
</div>
<div id="emailPreview">
    <pre>__ previewText __</pre>
</div>
```

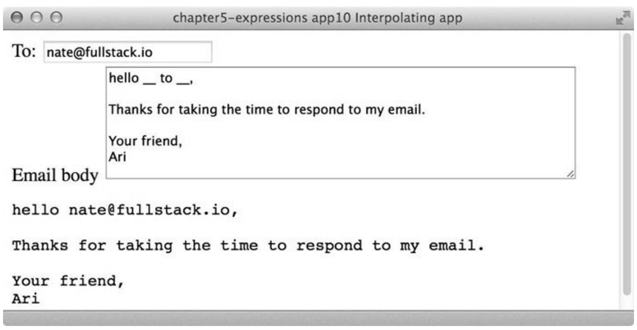


图6-1 插值文本

在线实例：http://jsbin.com/ivuJEXI/1/edit。

第 7 章

过滤器

过滤器用来格式化需要展示给用户的数据。AngularJS有很多实用的内置过滤器，同时也提供了方便的途径可以自己创建过滤器。

在HTML中的模板绑定符号{{ }}内通过|符号来调用过滤器。例如，假设我们希望将字符串转换成大写，可以对字符串中的每个字符都单独进行转换操作，也可以使用过滤器：

```
{{ name | uppercase }}
```

在JavaScript代码中可以通过`$filter`来调用过滤器。例如，在JavaScript代码中使用`lowercase`过滤器：

```
app.controller('DemoController', ['$scope', '$filter',
  function($scope, $filter) {

    $scope.name = $filter('lowercase')('Ari');
}]);
```

以HTML的形式使用过滤器时，如果需要传递参数给过滤器，只要在过滤器名字后面加冒号即可。如果有多个参数，可以在每个参数后面都加入冒号。例如，数值过滤器可以限制小数点后的位数，在过滤器后写上`:2`可以将2作为参数传给过滤器：

```
<!-- 显示：123.46 -->
{{ 123.456789 | number:2 }}
```

可以用|符号作为分割符来同时使用多个过滤器，后面介绍自定义过滤器时就会看到相关的例子。我们先来介绍AngularJS提供的内置过滤器。

1. currency

`currecy`过滤器可以将一个数值格式化为货币格式。用`{{ 123 | currency }}`来将123转化成货币格式。

`currecy`过滤器允许我们自己设置货币符号。默认情况下会采用客户端所处区域的货币符号，但是也可以自定义货币符号。

2. date

`date`过滤器可以将日期格式化成需要的格式。AngularJS中内置了几种日期格式，如果没有指定使用任何格式，默认会采用`mediumDate`格式，下面的例子中展示了这个格式。

下面是内置的支持本地化的日期格式：

```
{{ today | date:'medium' }}    <!-- Aug 09, 2013 12:09:02 PM -->
{{ today | date:'short' }}     <!-- 8/9/1312:09PM -->
{{ today | date:'fullDate' }}  <!-- Thursday, August 09, 2013 -->
{{ today | date:'longDate' }}  <!-- August 09, 2013 -->
{{ today | date:'mediumDate' }}<!-- Aug 09, 2013 -->
{{ today | date:'shortDate' }} <!-- 8/9/13 -->
{{ today | date:'mediumTime' }}<!-- 12:09:02 PM -->
{{ today | date:'shortTime' }} <!-- 12:09 PM -->
```

- 年份格式化

```
四位年份：{{ today | date:'yyyy' }} <!-- 2013 -->
两位年份：{{ today | date:'yy' }} <!-- 13 -->
一位年份：{{ today | date:'y' }} <!-- 2013 -->
```

- 月份格式化

```
英文月份：{{ today | date:'MMMM' }} <!-- August -->
英文月份简写：{{ today | date:'MMM' }} <!-- Aug -->
数字月份：{{ today |date:'MM' }} <!-- 08 -->
一年中的第几个月份：{{ today |date:'M' }} <!-- 8 -->
```

- 日期格式化

```
数字日期：{{ today|date:'dd' }} <!-- 09 -->
一个月中的第几天：{{ today | date:'d' }} <!-- 9 -->
英文星期：{{ today | date:'EEEE' }} <!-- Thursday -->
英文星期简写：{{ today | date:'EEE' }} <!-- Thu -->
```

- 小时格式化

```
24小时制数字小时：{{today|date:'HH'}} <!--00-->
一天中的第几个小时：{{today|date:'H'}} <!--0-->
12小时制数字小时：{{today|date:'hh'}} <!--12-->
上午或下午的第几个小时：{{today|date:'h'}} <!--12-->
```

- 分钟格式化

```
数字分钟数：{{ today | date:'mm' }} <!-- 09 -->
一个小时中的第几分钟：{{ today | date:'m' }} <!-- 9 -->
```

- 秒数格式化

```
数字秒数：{{ today | date:'ss' }} <!-- 02 -->
一分钟内的第几秒：{{ today | date:'s' }} <!-- 2 -->
毫秒数：{{ today | date:'.sss' }} <!-- .995 -->
```

- 字符格式化

```
上下午标识：{{ today | date:'a' }} <!-- AM -->
四位时区标识：{{ today | date:'Z' }} <!--- 0700 -->
```

下面是一些自定义日期格式的示例：

```
{{ today | date:'MMMd, y' }} <!-- Aug9, 2013 -->
{{ today | date:'EEEE, d, M' }} <!-- Thursday, 9, 8-->
{{ today | date:'hh:mm:ss.sss' }} <!-- 12:09:02.995 -->
```

3. filter

`filter`过滤器可以从给定数组中选择一个子集，并将其生成一个新数组返回。这个过滤器通

7

常用来过滤需要进行展示的元素。例如，在做客户端搜索时，可以从一个数组中立刻过滤出所需的结果。

这个过滤器的第一个参数可以是字符串、对象或是一个用来从数组中选择元素的函数。

下面分情况介绍传入不同类型的参数。

- 字符串

返回所有包含这个字符串的元素。如果我们想返回不包含该字符串的元素，在参数前加!符号。

- 对象

AngularJS会将待过滤对象的属性同这个对象中的同名属性进行比较，如果属性值是字符串就会判断是否包含该字符串。如果我们希望对全部属性都进行对比，可以将$当作键名。

- 函数

对每个元素都执行这个函数，返回非假值的元素会出现在新的数组中并返回。

例如，用下面的过滤器可以选择所有包含字母e的单词：

```
{{ ['Ari','Lerner','Likes','To','Eat','Pizza'] | filter:'e' }}
<!-- ["Lerner","Likes","Eat"] -->
```

如果要过滤对象，可以使用上面提到的对象过滤器。例如，如果有一个由people对象组成的数组，每个对象都含有他们最喜欢吃的食物的列表，那么可以用下面的形式进行过滤：

```
{{ [{
    'name': 'Ari',
    'City': 'San Francisco',
    'favorite food': 'Pizza'
    },{
    'name': 'Nate',
    'City': 'San Francisco',
    'favorite food': 'indian food'
    }] | filter:{'favorite food': 'Pizza'} }}
<!-- [{"name":"Ari","City":"SanFrancisco","favoritefood":"Pizza"}] -->
```

也可以用自定义函数进行过滤（在这个例子中函数定义在$scope上）：

```
{{ ['Ari','likes','to','travel'] | filter:isCapitalized }}
<!-- ["Ari"] -->
```

isCapitalized函数的功能是根据首字母是否为大写返回true或false，具体如下所示：

```
$scope.isCapitalized = function(str) {
    return str[0] == str[0].toUpperCase();
};
```

我们也可以给filter过滤器传入第二个参数，用来指定预期值同实际值进行比较的方式。

第二个参数可以是以下三种情况之一。

- true

用angular.equals(expected, actual)对两个值进行严格比较。

- false

进行区分大小写的子字符串比较。

- 函数

运行这个函数，如果返回真值就接受这个元素。

4. json

json过滤器可以将一个JSON或JavaScript对象转换成字符串。这种转换对调试非常有帮助：

```
{{ {'name': 'Ari', 'City': 'SanFrancisco'} | json }}
<!--
{
 "name": "Ari",
 "City": "San Francisco"
}
-->
```

5. limitTo

limitTo过滤器会根据传入的参数生成一个新的数组或字符串，新的数组或字符串的长度取决于传入的参数，通过传入参数的正负值来控制从前面还是从后面开始截取。

如果传入的长度值大于被操作数组或字符串的长度，那么整个数组或字符串都会被返回。

例如，我们可以截取字符串的前三个字符：

```
{{ San Francisco is very cloudy | limitTo:3 }}
<!-- San -->
```

或最后的六个字符：

```
{{ San Francisco is very cloudy | limitTo:-6 }}
<!-- cloudy -->
```

对数组也可以进行同样的操作。返回数组的第一个元素：

```
{{ ['a','b','c','d','e','f'] | limitTo:1 }}
<!-- ["a"] -->
```

6. lowercase

lowercase过滤器将字符串转为小写。

```
{{ "San Francisco is very cloudy" | lowercase }}
<!-- san francisco is very cloudy -->
```

7. number

number过滤器将数字格式化成文本。它的第二个参数是可选的，用来控制小数点后截取的位数。

如果传入了一个非数字字符，会返会空字符串。

```
{{ 123456789 | number }}
<!-- 1,234,567,890 -->
```

```
{{ 1.234567 | number:2 }}
<!-- 1.23 -->
```

8. orderBy

orderBy过滤器可以用表达式对指定的数组进行排序。

orderBy可以接受两个参数，第一个是必需的，第二个是可选的。

第一个参数是用来确定数组排序方向的谓词。

下面分情况讨论第一个参数的类型。

- 函数

当第一个参数是函数时，该函数会被当作待排序对象的getter方法。

- 字符串

对这个字符串进行解析的结果将决定数组元素的排序方向。我们可以传入+或-来强制进行升序或降序排列。

- 数组

在排序表达式中使用数组元素作为谓词。对于与表达式结果并不严格相等的每个元素，则使用第一个谓词。

第二个参数用来控制排序的方向（是否逆向）。

例如，我们将下面的对象数组用name字段进行排序：

```
{{ [{
    'name': 'Ari',
    'status': 'awake'
    },{
    'name': 'Q',
    'status': 'sleeping'
    },{
    'name': 'Nate',
    'status': 'awake'
    }] | orderBy:'name' }}
<!--
  [
  {"name":"Ari","status":"awake"},
  {"name":"Nate","status":"awake"},
  {"name":"Q","status":"sleeping"}
  ]
-->
```

也可以对排序结果进行反转。例如，通过将第二个参数设置为true可以将排序结果进行反转：

```
{{ [{
    'name': 'Ari',
    'status': 'awake'
    },{
    'name': 'Q',
    'status': 'sleeping'
    },{
    'name': 'Nate',
```

```
    'status': 'awake'
    }] | orderBy:'name':true }}
<!--
  [
  {"name":"Q","status":"sleeping"},
  {"name":"Nate","status":"awake"},
  {"name":"Ari","status":"awake"}
  ]
-->
```

9. **uppercase**

uppercase过滤器可以将字符串转换为大写形式：

```
{{ "San Francisco is very cloudy" | uppercase }}
<!-- SAN FRANCISCO IS VERY CLOUDY -->
```

7.1　自定义过滤器

正如前面所见，创建自定义过滤器非常容易。创建自定义过滤器需要将它放到自己的模块中。下面我们一起来实现一个过滤器，将字符串第一个字母转换为大写。

首先，创建一个模块用以在应用中进行引用（这是一个非常好的实践）：

```
angular.module('myApp.filters', [])
.filter('capitalize', function() {
  return function(input) {};
});
```

过滤器本质上是一个会把我们输入的内容当作参数传入进去的函数。上面这个例子中，我们在调用过滤器时简单地把input当作字符串来处理。可以在这个函数中做一些错误检查：

```
angular.module('myApp.filters', [])
.filter('capitalize', function() {
  return function(input) {
    // input是我们传入的字符串
    if (input) {
      return input[0].toUpperCase() + input.slice(1);
  }
});
```

现在，如果想将一个句子的首字母转换成大写形式，可以用过滤器先将整个句子都转换成小写，再把首字母转换成大写：

```
<!-- Ginger loves dog treats -->
{{ 'ginger loves dog treats' | lowercase | capitalize }}
```

7.2　表单验证

能够根据用户在表单中输入的内容给出实时视觉反馈是非常重要的。在人与人沟通的语境中，表单验证给出来的反馈同获得正确输入同等重要。

表单验证不仅能给用户提供有用的反馈，同时也能保护我们的Web应用不会被恶意或者错误的输入所破坏。我们要在Web前端尽力保护后端。

AngularJS能够将HTML5表单验证功能同它自己的验证指令结合起来使用，并且非常方便。

AngularJS提供了很多表单验证指令，我们会介绍其中一些核心的验证功能，然后介绍如何创建自己的验证器。

```
<form name="form" novalidate>
  <label name="email">Your email</label>
  <input type="email"
        name="email"
        ng-model="email" placeholder="Email Address" />
</form>
```

借助AngularJS，我们不需要花太多额外的精力就可以轻松实现客户端表单验证功能。虽然Web应用安全不能完全依赖客户端验证，但客户端验证可以提供表单状态的实时反馈。

要使用表单验证，首先要确保表单像上面的例子一样有一个name属性。

所有输入字段都可以进行基本的验证，比如最大、最小长度等。这些功能是由新的HTML5表单属性提供的。

如果想要屏蔽浏览器对表单的默认验证行为，可以在表单元素上添加novalidate标记。

下面看一下可以在input元素上使用的所有验证选项。

1. 必填项

验证某个表单输入是否已填写，只要在输入字段元素上添加HTML5标记required即可：

```
<input type="text" required />
```

2. 最小长度

验证表单输入的文本长度是否大于某个最小值，在输入字段上使用AngularJS指令ng-minleng="{number}"：

```
<input type="text" ng-minlength="5" />
```

3. 最大长度

验证表单输入的文本长度是否小于或等于某个最大值，在输入字段上使用AngularJS指令ng-maxlength="{number}"：

```
<input type="text" ng-maxlength="20" />
```

4. 模式匹配

使用ng-pattern="/PATTERN/"来确保输入能够匹配指定的正则表达式：

```
<input type="text" ng-pattern="[a-zA-Z]" />
```

5. 电子邮件

验证输入内容是否是电子邮件，只要像下面这样将input的类型设置为email即可：

```
<input type="email" name="email" ng-model="user.email" />
```

6. 数字

验证输入内容是否是数字，将input的类型设置为number：

```
<input type="number" name="age" ng-model="user.age" />
```

7. URL

验证输入内容是否是URL，将input的类型设置为url：

```
<input type="url" name="homepage" ng-model="user.facebook_url" />
```

8. 自定义验证

在AngularJS中自定义指令是非常容易的。鉴于目前还没有介绍到指令的相关内容，第10章再深入研究如何创建自定义验证。目前先来看一下如何通过向后端服务器发送请求，并通过响应的结果来将输入字段设置为合法或不合法，以确保输入字段中的内容是唯一的。

9. 在表单中控制变量

表单的属性可以在其所属的$scope对象中访问到，而我们又可以访问$scope对象，因此JavaScript可以间接地访问DOM中的表单属性。借助这些属性，我们可以对表单做出实时（和AngularJS中其他东西一样）响应。这些属性包括下面这些。

（注意，可以使用下面的格式访问这些属性。）

```
formName.inputFieldName.property
```

- 未修改的表单

这是一个布尔属性，用来判断用户是否修改了表单。如果未修改，值为true，如果修改过值为false：

```
formName.inputFieldName.$pristine
```

- 修改过的表单

只要用户修改过表单，无论输入是否通过验证，该值都返回true：

```
formName.inputFieldName.$dirty
```

- 合法的表单

这个布尔型的属性用来判断表单的内容是否合法。如果当前表单内容是合法的，下面属性的值就是true：

```
formName.inputFieldName.$valid
```

- 不合法的表单

这个布尔属性用来判断表单的内容是否不合法。如果当前表单内容是不合法的，下面属性的值为true：

```
formName.inputFieldName.$invalid
```

- 错误

这是AngularJS提供的另外一个非常有用的属性：$error对象。它包含当前表单的所有验证内容，以及它们是否合法的信息。用下面的语法访问这个属性：

```
formName.inputfieldName.$error
```

如果验证失败，这个属性的值为true；如果值为false，说明输入字段的值通过了验证。

10. 一些有用的CSS样式

AngularJS处理表单时，会根据表单当前的状态添加一些CSS类（例如当前是合法的、未发生变化的，等等），这些CSS类的命名和前面介绍的属性很相似。

它们包括：

```
.ng-pristine {}
.ng-dirty {}
.ng-valid {}
.ng-invalid {}
```

它们对应着表单输入字段的特定状态。

当某个字段中的输入非法时，.ng-invlid类会被添加到这个字段上。当前例子中的站点将对应的CSS样式设置为：

```
input.ng-invalid {
  border: 1px solid red;
}
input.ng-valid {
  border: 1px solid green;
}
```

- $parsers

当用户同控制器进行交互，并且ngModelController中的$setViewValue()方法被调用时，$parsers数组中的函数会以流水线的形式被逐个调用。第一个$parse被调用后，执行结果会传递给第二个$parse，以此类推。

这些函数可以对输入值进行转换，或者通过$setValidity()函数设置表单的合法性。

使用$parsers数组是实现自定义验证的途径之一。例如，假设我们想要确保输入值在某两个数值之间，可以在$parsers数组中入栈一个新的函数，这个函数会在验证链中被调用。

每个$parser返回的值都会被传入下一个$parser中。当不希望数据模型发生更新时返回undefined。

```
angular.module('myApp')
.directive('oneToTen', function() {
   return {
     require: '?ngModel',
     link: function(scope, ele, attrs, ngModel) {
       if (!ngModel) return;
         ngModel.$parsers.unshift(
           function(viewValue) {
             var i = parseInt(viewValue);

             if (i >= 0 && i < 10) {
               ngModel.$setValidity('oneToTen', true);
               return viewValue;
             } else {
                 ngModel.$setValidity('oneToTen', false);
                 return undefined;
             }
```

```
            });
        }
    };
});
```

- $formatters

当绑定的ngModel值发生了变化，并经过$parsers数组中解析器的处理后，这个值会被传递给$formatters流水线。同$parsers数组可以修改表单的合法性状态类似，$formatters中的函数也可以修改并格式化这些值。

比起单纯的验证目的，这些函数更常用来处理视图中的可视变化。例如，假设我们要对某个值进行格式化。通过$formatters数组可以在这个值上执行过滤器：

```
angular.module('myApp')
.directive('oneToTen', function() {
    return {
        require: '?ngModel',
        link: function(scope, ele, attrs, ngModel) {
            if (!ngModel) return;

            ngModel.$formatters.unshift(function(v) {
                return $filter('number')(v);
            });
        }
    };
});
```

11. 组合实例

下面我们一起创建一个注册表单。表单中包括用户的名字、邮件地址以及用户名。

开始之前，首先看一下我们希望这个表单长成什么样子，如图7-1所示。

在线示例：http://jsbin.com/ePomUnI/5/edit。

下面开始定义表单：

```
<form name="signup_form" novalidateng-submit="signupForm()">
    <fieldset>
    <legend>Signup</legend>

    <button type="submit" class="button radius">Submit</button>
   </fieldset>
</form>
```

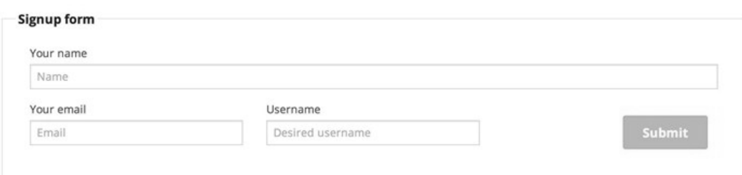


图7-1　注册表单

这个表单的名称是signup_form，当表单提交时我们要调用signupForm()。

下面添加用户的名字：

```
<div class="row">
    <div class="large-12 columns">
        <label>Your name</label>
        <input type="text"
            placeholder="Name"
            name="name"
            ng-model="signup.name"
            ng-minlength="3"
            ng-maxlength="20" required />
    </div>
</div>
```

后面的章节会讨论样式方面的内容，现在我们只会简单地把样式引入进来。本章使用Foundation[①]作为CSS布局的框架。

我们添加了一个表单，这个表单有一个名为name的输入字段，并且这个输入字段被ng-model指令绑定到了$scope对象的signup.name上。

> 不要忘记给输入字段添加name属性。给输入字段添加name属性非常重要：这决定了我们将验证信息展示给用户时如何引用表单输入字段。

同时，我们也设置了一些验证。验证要求name字段的最小长度是3个字符，最大长度是20个字符（当长度大于等于21时输入会变为不合法）。最后，我们要求name字段是必填项。

通过使用这些属性，可以在表单未通过验证时控制展示或隐藏错误列表。用$dirty属性来确保用户未对输入内容进行修改时错误内容不会显示出来：

```
<divclass="row">
    <div class="large-12 columns">
        <label>Your name</label>
        <input type="text"
            placeholder="Name"
            name="name"
            ng-model="signup.name"
            ng-minlength="3"
            ng-maxlength="20" required />
        <div class="error"
            ng-show="signup_form.name.$dirty && signup_form.name.$invalid">
            <small class="error"
                ng-show="signup_form.name.$error.required">
                Your name is required.
            </small>
            <small class="error"
                ng-show="signup_form.name.$error.minlength">
                Your name is required to be at least 3 characters
            </small>
            <small class="error"
                ng-show="signup_form.name.$error.maxlength">
                Your name cannot be longer than 20 characters
            </small>
        </div>
    </div>
</div>
```

① http://foundation.zurb.com/

将整个过程分开来看，我们只是像以前一样在表单发生改变，且输入内容不合法时才展示错误内容。现在，我们会在特定的属性未通过验证时只展示对应的特定DOM元素。

接下来看下一组验证，电子邮箱的验证：

```
<div class="row">
    <div class="large-12 columns">
        <label>Your email</label>
        <input type="email"
            placeholder="Email"
            name="email"
            ng-model="signup.email"
            ng-minlength="3" ng-maxlength="20" required />
        <div class="error"
            ng-show="signup_form.email.$dirty && signup_form.email.$invalid">
            <small class="error"
                ng-show="signup_form.email.$error.required">
                Your email is required.
            </small>
            <small class="error"
                ng-show="signup_form.email.$error.minlength">
                Your email is required to be at least 3 characters
            </small>
            <small class="error"
                ng-show="signup_form.email.$error.email">
                That is not a valid email. Please input a valid email.
            </small>
            <small class="error"
                ng-show="signup_form.email.$error.maxlength">
                Your email cannot be longer than 20 characters
            </small>
        </div>
    </div>
</div>
```


现在整个表单都被包含进来了，我们来看一下电子邮件的输入字段。注意，我们将输入字段的`type`属性设置为`email`，并且在`$error.email`上添加了验证错误的信息。这个验证同时基于AngularJS和HTML5属性实现。

最后，看一下用户名的输入字段：

```
<div class="large-12 columns">
    <label>Username</label>
    <input type="text"
        placeholder="Desired username"
        name="username"
        ng-model="signup.username"
        ng-minlength="3"
        ng-maxlength="20"
        ensure-unique="username" required />
    <div class="error"
        ng-show="signup_form.username.$dirty &&
                 signup_form.username.$invalid">
        <small class="error"
            ng-show="signup_form.username.$error.required">
            Please input a username
        </small>
        <small class="error"
            ng-show="signup_form.username.$error.minlength">
```

```
            Your username is required to be at least 3 characters
        </small>
        <small class="error"
            ng-show="signup_form.username.$error.maxlength">
            Your username cannot be longer than 20 characters
        </small>
        <small class="error"
            ng-show="signup_form.username.$error.unique">
            That username is taken, please try another
        </small>
    </div>
</div>
```

在最后一个输入字段中除了同前面相同的验证外，还添加了一个自定义验证。这个自定义验证是用AngularJS指令定义的：

```
app.directive('ensureUnique', function($http) {
    return {
        require: 'ngModel',
        link: function(scope, ele, attrs, c) {
            scope.$watch(attrs.ngModel, function(n) {
                if (!n) return;
                $http({
                    method: 'POST',
                    url: '/api/check/' + attrs.ensureUnique,
                    data: {
                    field: attrs.ensureUnique,
                    value: scope.ngModel
                  }
                }).success(function(data) {
                    c.$setValidity('unique', data.isUnique);
                }).error(function(data) {
                    c.$setValidity('unique', false);
                });
            });
        }
    };
});
```

当表单内容通过验证后，会向/api/check/username发送一个POST请求来验证用户名是否可用。显然由于我们一直在讨论前端的代码，现在并没有可以用来测试这些内容的后端，尽管实现这个后端并不复杂。

最后，把按钮放进去。可以用ng-disabled指令基于表单的合法性来启用或禁用按钮：

```
<button type="submit"
    ng-disabled="signup_form.$invalid"
    class="button radius">Submit</button>
```

在线示例：http://jsbin.com/ePomUnI/5/edit。

之前提到过，表单本身会提供$invalid和$valid属性。

尽管实时验证非常有用，但是当用户还没有完成输入时就弹出一个错误提示，这种体验是非常糟糕的。应该在用户提交表单或完成当前字段中的输入后，再提示验证信息，这样才是用户友好的。下面看看如何实现这两种效果。

- 在提交后显示验证信息

当用户试图提交表单时，你可以在作用域中捕获到一个submitted值，然后对表单内容进行验证并显示错误信息。

例如，修改一下前面的例子，只在用户提交表单时才显示错误信息。在ng-show指令中加入对表单是否进行了提交的检查（后面会实现这个功能）：

```
<form name="signup_form"
    novalidate
    ng-submit="signupForm()"
    ng-controller="signupController">
    <fieldset>
        <legend>Signup</legend>
        <div class="row">
            <div class="large-12 columns">
                <label>Your name</label>
                <input type="text"
                    placeholder="Name"
                    name="name"
                    ng-model="signup.name"
                    ng-minlength="3"
                    ng-maxlength="20" required />
                <div class="error"
                    ng-show="signup_form.name.$dirty && signup_form.name.$invalid &&
                    signup_form.submitted">
                    <small class="error"
                        ng-show="signup_form.name.$error.required">
                        Your name is required.
                    </small>
                    <small class="error"
                        ng-show="signup_form.name.$error.minlength">
                        Your name is required to be at least 3 characters
                    </small>
                    <small class="error"
                        ng-show="signup_form.name.$error.maxlength">
                        Your name cannot be longer than 20 characters
                    </small>
                </div>
            </div>
        </div>
        <button type="submit" >Submit</button>
    </fieldset>
</form>
```

现在，仅当signup_form.submitted设置为true时，容纳错误信息的div才会展示出来。在signupForm操作中实现这个行为，如下所示：

```
app.controller('signupController', function($scope) {
    $scope.submitted = false;
    $scope.signupForm = function() {
        if ($scope.signup_form.$valid) {
            // 正常提交
        } else {
            $scope.signup_form.submitted = true;
        }
    }
});
```

如果用户试图在有非法输入的情况下提交表单，我们现在可以捕获到这个行为并展示合适的错误信息。

在线示例：http://jsbin.com/ePomUnI/6/edit。

● *在失焦后显示验证信息*

如果想保留实时错误提示的体验，可以在用户从某个输入字段失焦后提示错误信息（例如用户已经不在某个特定的输入字段中时）。为了实现这个效果，需要实现一个不是很复杂的指令，并向表单中添加一个新的变量。

我们需要使用的指令是ngFocus，它是这样的：

```
app.directive('ngFocus', [function() {
    var FOCUS_CLASS = "ng-focused";
    return {
        restrict: 'A',
        require: 'ngModel',
        link: function(scope, element, attrs, ctrl) {
            ctrl.$focused = false;
            element.bind('focus', function(evt) {
                element.addClass(FOCUS_CLASS);
                scope.$apply(function() {
                    ctrl.$focused = true;
                });
            }).bind('blur', function(evt) {
                element.removeClass(FOCUS_CLASS);
                scope.$apply(function() {
                    ctrl.$focused = false;
                });
            });
        }
    };
}]);
```

将ngFocus指令添加到input元素上就可以使用这个指令，如下所示：

```
<input ng-class="{error: signup_form.name.$dirty && signup_form.name.$invalid}"
    type="text"
    placeholder="Name"
    name="name"
    ng-model="signup.name"
    ng-minlength="3"
    ng-maxlength="20" required ng-focus />
```

ngFocus指令给表单输入字段的blur和focus添加了对应的行为，添加了一个名为ng-focused的类，并将$focused的值设置为true。接下来，可以根据表单是否具有焦点来展示独立的错误信息。如下所示：

```
<div class="error"
    ng-show="signup_form.name.$dirty && signup_form.name.$invalid && !signup_form.name.$focused">
```

在线示例：http://jsbin.com/ePomUnI/7/edit。

也可以在ngModel控制器中使用$isEmpty()方法来判断输入字段是否为空。当输入字段为空这个方法会返回true，反之如果不为空则返回false。

ngMessages(1.3+)

众所周知，表单和验证是Angular中复杂的组件之一。上面的例子不是特别好，不简洁。在Angular 1.3发布前，表单验证必须以这种方式编写。

然而在发布的Angular 1.3中，Angular核心做了一个升级。它不再需要基于一个详细的表达式状态创建元素显示或隐藏（正如我们在本章所做的那样）。

```
<form name="signup_form" novalidate ng-submit="signupForm()"
ng-controller="signupController">
    <fieldset>
        <legend>Signup</legend>
        <div class="row">
            <div class="large-12 columns">
                <label>Your name</label>
                <input type="text" placeholder="Name" name="name" ng-model="signup.name"
                    ng-minlength=3 ng-maxlength=20 required />
                <div class="error" ng-show="signup_form.name.$dirty && signup_form.name.
                    $invalid && signup_form.submitted">
                    <small class="error" ng-show="signup_form.name.$error.required">
                        Your name is required.</small>
                    <small class="error" ng-show="signup_form.name.$error.minlength">
                        Your name is required to be at least 3 characters</small>
                    <small class="error" ng-show="signup_form.name.$error.maxlength">
                        Your name cannot be longer than 20 characters </small>
                </div>
            </div>
        </div>
        <button type="submit">Submit</button>
    </fieldset>
</form>
```

本质上这一功能会检查错误对象的状态发生了变化。此外，我们还得到了站点中每个表单需要的很多额外的和重复的标记。这显然不是一个理想的解决方案。

从1.3开始，Angular中新增了一个ngMessages指令。

安装

安装ngMessages很简单，因为它被打包成了一个Angular模块。首先下载这个模块：

```
$ bower install --save angular-messages
```

或者，也可以从angular.org下载该文件并将它保存到项目中。还需要将angular-messages.js这个JavaScript引入我们的主HTML中：

```
<script type="text/javascript" src="bower_components/angular-messages/angular-messages.js">
    </script>
```

最后，我们还要告诉Angular将ngMessages作为应用程序的依赖模块引入，就像这样：

```
angular.module('myApp', ['ngMessages']);
```

现在，我们已经安装了ngMessages，然后可以马上开始使用它了。使用前面的例子作为基础，你可以移除ng-show指令，然后使用ngMessages的一个更简洁的实现替换它。

```
<form name="signup_form" novalidate ng-submit="signupForm()"
ng-controller="signupController">
```

```
    <label>Your name</label>
    <input type="text" placeholder="Name" name="name" ng-model="signup.name" ng-minlength=
        3 ng-maxlength=20 required />
    <div class="error" ng-messages="signup_form.name.$error">
        <div ng-message="required">Make sure you enter your name</div>
        <div ng-message="minlength">Your name must be at least 3 characters</div>
        <div ng-message="maxlength">Your name cannot be longer than 20 characters</div>
    </div>
    <button type="submit">Submit</button>
</form>
```

借助ngMessages，表本身比前面的实现更清洁，并且更好理解。

然而对于这个实现，一次只会显示一个错误消息。如果我们想要更新这个实现同时显示所有的错误将会怎样？很容易。只需在ng-message指令旁边使用ng-messages-multiple属性即可。

```
<div class="error" ng-messages="signup_form.name.$error" ng-messages-multiple>
    <div ng-message="required"> sure you enter your name</div>
    <div ng-message="minlength">Your name must be at least 3 characters</div>
    <div ng-message="maxlength">Your name cannot be longer than 20 characters</div>
</div>
```

很多时候这些信息相互之间非常相似。我们可以将它们保存到模板中从而减少麻烦，而不是重新输入每个字段的错误信息。

```
<!-- In templates/errors.html -->
<div ng-message="required">This field is required</div>
<div ng-message="minlength">The field must be at least 3 characters</div>
<div ng-message="maxlength">The field cannot be longer than 20 characters</div>
```

然后我们可以通过在视图中使用ng-messages-include属性引入这个模板来改进这个表单：

```
<div class+'error' ng-messages="signup_form.name.$error"
ng-messages-include="templates/errors.html">
</div>
```

有时，你可能希望为不同的字段自定义错误信息。没问题，你可以在这个指令内简单地插入一个自定义错误信息。由于ngMessages涉及ngMessages容器中错误列表的顺序，我们可以通过在这个指令中列出自定义错误信息的方式覆盖它们。

```
<div class="error" ng-messages="signup_form.name.$error"
ng-messages-include="templates/errors.html">
<!--
除了minlength会被覆盖之外，其他每个信息都会保持不变
-->
</div>
```

此外，甚至还可以为自定义验证创建自定义消息。可以通过修改模型的$parsers链做到这一点。

例如，比方说我们想要创建一个自定义验证器验证用户名在一个注册表单中是否有效：

```
app.directive('ensureUnipue', function($http) {
    return {
        require: 'ngModel',
        link: function(scope, ele, attrs, ctrl) {

            ctrl.$parsers.push(function(val) {
```

```
                // 在这里添加验证
            });
        }
    }
});
```

对于ngModel，你可以添加可以使用ngMessage指令显示/隐藏的自定义信息。还可以添加可以使用ngMessage指令检查的带有自定义的消息的指令。例如，改变前面使用ngMessages的例子。

```
<form name="signup_form" novalidate ng-submit="signupForm()" ng-controller="signupController"
ensure-unique="/api/checkUsername.json">
    <label>
        Your name
    </label>
    <input type="text" placeholder="Username" name="username" ng-model="signup.username"
    ng-minlength=3 ng-maxlength=20 required />
    <div class="error" ng-messages="signup_form.username.$error">
        <div ng-message="required">
            Make sure you enter your username
        </div>
        <div ng-message="checkingAvailability">
            Checking...
        </div>
        <div ng-message="usernameAvailablity">
            The username has already been taken. Please choose another
        </div>
    </div>
    <button type="submit">
        Submit
    </button>
</form>
```

在这中用法中，我们检查了错误信息的自定义属性。为了添加自定义错误消息，我们将会把它们应用到自定义ensureUnique指令的ngModel中。

```
app.directive('ensureUnique', function($http) {
    return {
        require: 'ngModel',
        link: function(scope, ele, attrs, ctrl) {
            var url = attrs.ensureUnique;

            ctrl.$parsers.push(function(val) {
                if (!val || val.length === 0) {
                    return;
                }

                ngModel.$setValidity('checkingAvailability', true);
                ngModel.$setValidity('usernameAvailablity', false);

                $http({
                    method: 'GET',
                    url: url,
                    params: {
                        username: val
                    }
                }).success(function() {
                    ngModel
                        .$setValidity('checkingAvailability', false);
                    ngModel
                        .$setValidity('usernameAvailablity', true);
```

7

```
                })['catch'](function() {
                    ngModel
                        .$setValidity('checkingAvailability', false);
                    ngModel
                        .$setValidity('usernameAvailablity', false);
                });
                return val;
            })
        }
    }
});
```

第 8 章

指令简介

8

作为Web开发者我们都非常熟悉HTML。下面简单回顾一下并统一我们对这个最基本的Web技术的认识。

1. HTML文档

HTML文档是一个纯文本文件，包含了页面的结构以及由CSS定义的样式，或者可以操作样式的JavaScript代码。

2. HTML节点

HTML节点是嵌套在另一个元素内的元素或一串字符。除了文本节点外，所有元素都是节点。

3. HTML元素

HTML元素由一个开始标签和一个结束标签组成。

4. HTML标签

HTML标签用来标记元素的开始和结束。标签本身用尖括号来声明。

开始标记的名字会同时被当作元素的名字，同时标签还会包含用来修饰元素的属性。

5. 属性

属性用来给HTML元素添加额外的信息。这些属性设置在开始标记中。可以使用形如`key="value"`的键值对设置它们，或者只设置键。

我们看看`<a>`超链接标签，它可以创建从一个页面到另一个页面的链接：很多标签和超链接标签一样，会有很多特殊的属性，这些属性就好比标签的参数。例如，超链接标签的`href`属性会激活该标签的行为，同时在大多数浏览器中会将开始和结束标记中间的文本转换为默认的蓝色。

```
<a href="http://google.com">
    Click me to go to Google</a>
```

`<a>`标签定义了一个从当前页面到本站或站外另一个页面之间的链接，`href`属性定义了链接的目标。

而下面这个按钮元素则与此非常不同：

```
<button href="http://google.com"
    type="submit">Click me</button>
```

默认情况下超链接标签是蓝色且有下划线的，而按钮标签在浏览器中看起来是一个可点击的按钮。

超链接标签知道当自己的href属性被设置为http://google.com之后，如果用户点击这个超链接，它应该修改地址栏的URL并加载Google的首页。

而按钮标签则完全忽略href属性，并不会在被点击时有同样的行为。

因此，修改地址栏的URL并将你带到一个新的页面是超链接的预置行为，而不是按钮的预置行为。

最后，两个标签在设置了title属性时则有相同的行为：当用户将鼠标悬停在元素上时会出现一个提示框。

```
<a href="http://google.com"
    title="click me">
    Click me to go to Google
</a>
<button type="submit"
    title="click me">Click me</button>
```

总地来说，浏览器会渲染HTML元素的样式和行为，这个能力是Web强大功能的基础之一。

任何一个浏览器厂商，无论是Google或Microsoft都尽量遵循同样的HTML标准，以此来保证Web编程在跨设备和操作系统时的一致性。

老版本的IE浏览器并没有遵循标准的HTML定义，因此我们需要一些技巧才能让其正常工作。更多内容请查看第30章。

近来出现了很多新的HTML标签，它们是HTML5标准的一部分。例如video标签可以定义一个视频、电影剪辑或流视频：

```
<video href="/goofy-video.mp4"></video>
```

这些HTML5的新标签在比较新的浏览器中可以正常工作，但是IE8或更早的IE浏览器都没有对其提供支持。

8.1　指令：自定义 HTML 元素和属性

基于我们对HTML元素的理解，指令本质上就是AngularJS扩展具有自定义功能的HTML元素的途径。例如，我们可以创建一个自定义元素，它实现了<video>标签的功能并且能在所有浏览器中工作：

```
<my-better-video my-href="/goofy-video.mp4">
Caneventaketext</my-better-video>
```

注意，这个自定义元素使用了特殊的开始和闭合标签my-better-video，以及my-href这个自定义属性。

为了让这个标签更有用，可以将浏览器默认的video标签重载，用下面这种写法代替它：

```
<video my-href="/goofy-video.mp">
    Can still take children nodes
</video>
```

正如我们看到的那样，指令可以和其他指令或属性组合在一起使用，这种组合使用的方式叫

做合成。

为了有效了解如何将一个个小组件组合成一个复杂的系统，首先要了解更基础的内容。接下来几节的目标就是帮助你了解这些基础内容，我们开始吧。

1. HTML引导

当浏览器加载一个包含AngularJS应用的HTML时，我们只需要一小段很简单的代码就能够启动AngularJS应用（前面的章节介绍过相关内容）。

在HTML中要用内置指令ng-app标记出应用的根节点。这个指令需要以属性的形式来使用，因此可以将它写到任何位置，但是写到`<html>`的开始标签上是最常规的做法：

> 内置指令是打包在AngularJS内部的指令。所有内置指令的命名空间都使用ng作为前缀。为了防止命名空间冲突，不要在自定义指令前加ng前缀。

```
<html ng-app="myApp">
    <!-- 应用的$rootScope -->
</html>
```

现在，在HTML元素中可以使用所有内置或自定义指令了。同时，基于JavaScript的原型继承机制，任何在这个根元素内部的指令只要能够访问作用域，就可以访问`$rootScope`。这里的能够访问作用域指的是同DOM进行了链接，这个操作会在指令稍后的生命周期中进行。

由于指令的生命周期非常复杂，会有专门的章节来介绍。在那部分内容中还会讨论指令中哪些方法是可以访问作用域的，以及作用域是如何在指令间共享的。详细内容请查看第10章。

2. 我们的第一个指令

学习指令最快的途径就是亲自使用它，让我们来创建一个自定义指令。看看下面的HTML元素，后面我们会用到它：

```
<my-directive></my-directive>
```

假设我们已经创建了一个完整的HTML文档，其中包含了AngularJS，并且DOM中已经用ng-app指令标识出了应用的根元素，当AngularJS编译HTML时就会调用指令。

> 10.3节会介绍指令生命周期中的编译阶段。

调用指令意味着执行指令背后与之相关联的JavaScript代码，这些代码是我们用指令定义写出来的。

`myDirective`指令的定义看起来是这样的：

```
angular.module('myApp',[])
.directive('myDirective', function() {
    return {
        restrict: 'E',
        template: '<a href="http://google.com">
        Click me to go to Google</a>'
    };
});
```

上面的JavaScript代码就是指令定义，效果如图8-1所示。10.1节会介绍定义指令时所有的可用设置。

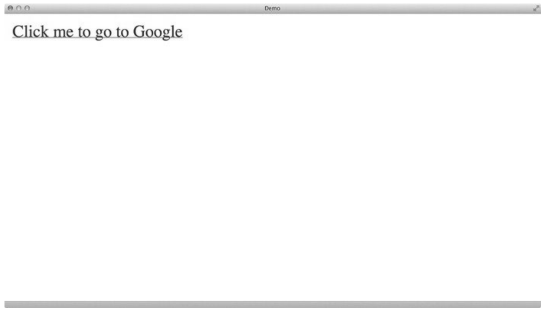


图8-1　简单指令的实践

通过AngularJS模块API中的`.directive()`方法，我们可以通过传入一个字符串和一个函数来注册一个新指令。其中字符串是这个指令的名字，指令名应该是驼峰命名风格的，函数应该返回一个对象。

驼峰命名风格用来将一个短语写在一个单词中，除了第一个单词外其他单词首字母大写，中间不加空格。例如，bumpy roads用驼峰风格来写应该是bumpyRoads。

在我们的例子中，在HTML里使用`my-directive`声明指令，因此指令定义必须以`myDirective`为名字。

`directive()`方法返回的对象中包含了用来定义和配置指令所需的方法和属性。

为了尽快掌握简单的属性定义，我们只用了`restrict`和`template`两个设置项来定义指令。

第10章将详细介绍定义指令时所有可用的方法和属性，但现在，先用Google Chrome的开发者工具来对比一下输入的HTML和输出的HTML。

首先用Chrome打开你的HTML文档，会看到一个蓝色的“Click Here”链接。点击View→Developer→View Source来查看源代码，会看见图8-2所示的画面。

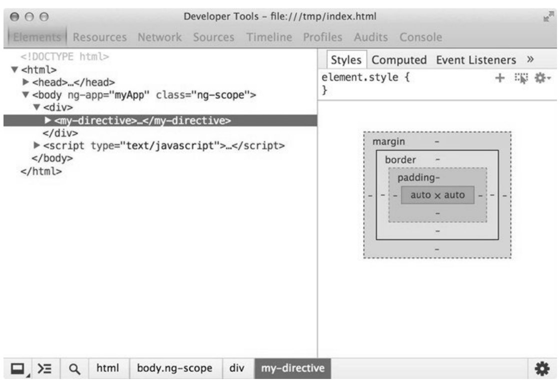


图8-2　Chrome开发者工具

注意，代码和你在文本编辑器中输入的没有区别，同时其中并没有一个链接标签。但的确屏幕上有一个链接写着“Click Here”，这是怎么回事？

为了分析这个现象，右键点击链接，在弹出菜单中选择Inspect Element，如图8-3所示。

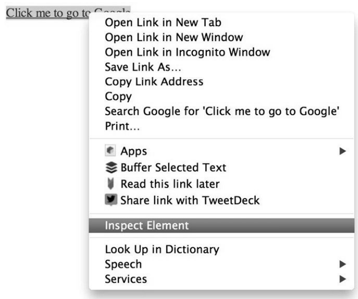


图8-3　Inspect Element

这样就可以打开Chrome开发者工具，并看到AngularJS在页面加载以及调用指令定义后生成的代码，AngularJS把生成后的代码提供给Chrome进行渲染，如图8-4所示。

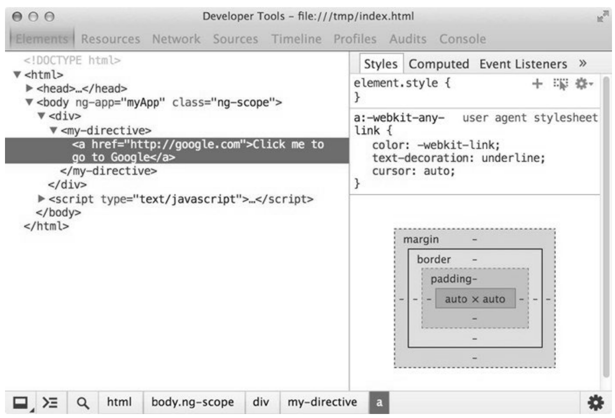


图8-4　查看指令内部

默认情况下，AngularJS将模板生成的HTML代码嵌套在自定义标签`<my-directive>`内部。

下面向指令定义中添加一些新的设置：我们可以将自定义标签从生成的DOM中完全移除掉，并只留下由模版生成的链接。将`replace`设置为`true`就可以实现这个效果：

```
angular.module('myApp', [])
.directive('myDirective', function() {
    return {
        restrict: 'E',
        replace: true,
        template: '<a href="http://google.com">Click me to go to Google</a>'
    };
});
```

再次看一下生成后的代码，会发现DOM中原始的指令声明已经不见了，只有我们在模板中写的HTML代码。`replace`方法会用自定义元素取代指令声明，而不是嵌套在其内部，如图8-5所示。

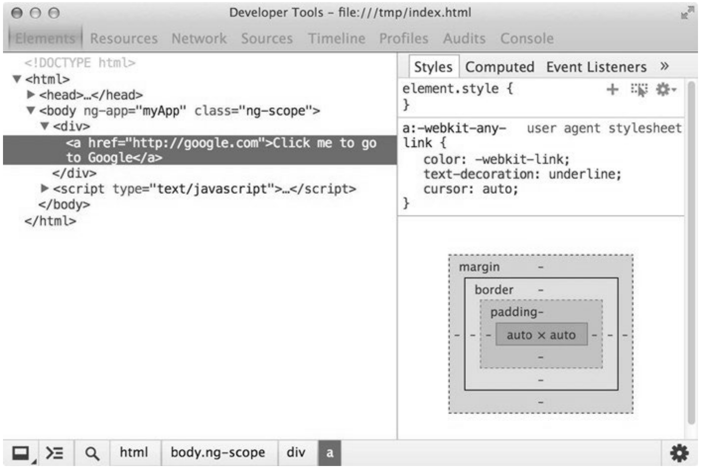


图8-5　替换现有元素

从现在起，我们把创建的这些自定义元素称作指令（用.directive()方法创建），因为事实上声明指令并不需要创建一个新的自定义元素。

声明指令本质上是在HTML中通过元素、属性、类或注释来添加功能。

下面都是用来声明前面创建指令的合法格式：

```
<my-directive></my-directive>
<div my-directive></div>
<div class="my-directive"></div>
<!--directive:my-directive-->
```

为了让AngularJS能够调用我们的指令，需要修改指令定义中的restrict设置。这个设置告诉AngularJS在编译HTML时用哪种声明格式来匹配指令定义。我们可以指定一个或多个格式。

例如，之前创建的指令中可以指定以元素（E）、属性（A）、类（C）或注释（M）的格式来调用指令：

```
angular.module('myApp', [])
.directive('myDirective', function() {
    return {
        restrict: 'EAC',
        replace: true,
        template: '<a href="http://google.com">Click me to go to Google</a>'
    };
});
```

无论有多少种方式可以声明指令，我们坚持使用属性方式，因为它有比较好的跨浏览器兼容性：

```
<div my-directive></div>
```

为了更加明确我们的意图，将restrict设置为字母A（代表attribute）：

```
restrict: 'A'
```

遵循这个约定的同时，也要注意每个浏览器的内置样式，以便决定指令模板在HTML中是嵌套在声明元素内，还是替换声明元素。

3. 关于IE浏览器

如果你正在使用IE浏览器，打开这个在线示例[①]。会发现尽管指令声明了两次，但只出现了一个链接。

从技术上讲，可以通过在文档头部声明新的标签（查看第30章）来解决这个问题，但这样做的后果就是，当疏忽了一致性时会导致额外的问题。

因此，好的经验法则就是始终用属性来声明指令（就像我们之前做的那样），这样会给以后带来方便。

值得注意的一个例外是，扩展内置HTML标签，例如用AngularJS重载`<a>`、`<form>`和`<input>`。这些场景不会导致浏览器的兼容性问题，因为它们本身就是浏览器所支持的标签。

4. 表达式

由于指令可以用属性的形式调用，我们可能会好奇如果给属性赋值会发生什么：

```
<h1 ng-init="greeting='HelloWorld'">
    The greeting is: {{ greeting }}
</h1>
```

在线示例：http://jsbin.com/IdUYexO/2/edit。

我们将表达式`greeting = 'Hello World'`赋值给内置指令`ng-init`。在表达式中，我们将`greeting`属性的值设置为`Hello World`，然后计算花括号内的`{{ greeting }}`这个表达式的值。

这两种情况都会在当前作用域中计算一个普通的JavaScript表达式。根据这个表达式放置的位置不同，当前作用域可以是AngularJS在应用启动时调用`ng-app`实例化的`$rootScope`，也可以是某个子作用域，比如某个控制器的作用域。

- 用表达式来声明指令

我们知道声明指令时既可以使用表达式，也可以不使用表达式。下面回顾一下几种合法的表达式声明：

```
<my-directive="someExpression">
</my-directive>
<div my-directive="someExpression">
</div>
<div class="my-directive:someExpression">
</div>
<!-- directive: my-directive someExpression -->
```

这里有一个值得注意的问题，赋值给指令的表达式会在哪个环境中运行？要回答这个问题，首先要了解一个复杂但非常重要的概念，就是当前作用域，它由DOM周围嵌套的控制器提供。

- 当前作用域介绍

首先快速了解一下由DOM通过内置指令`ng-controller`提供的作用域。这个指令的作用是在DOM中创建一个新的子作用域：

```
<p>We can access: {{ rootProperty }}</p>
```

① http://jsbin.com/IJAzUJE/1/edit

```
<div ng-controller="ParentController">
    <p>We can access: {{ rootProperty }}
    and {{ parentProperty }}</p>
    <div ng-controller="ChildController">
    <p>
        We can access:
        {{ rootProperty }} and
        {{ parentProperty }} and
        {{ childProperty }}
    </p>
    <p>{{ fullSentenceFromChild }}</p>
</div>
</div>

angular.module('myApp', [])
.run(function($rootScope) {
    // 使用.run访问$rootScope
    $rootScope.rootProperty = 'root scope';
})
.controller('ParentController', function($scope) {
    // 使用.controller访问`ng-controller`内部的属性
    // 在DOM忽略的$scope中，根据当前控制器进行推断
    $scope.parentProperty = 'parent scope';
})
.controller('ChildController', function($scope) {
    $scope.childProperty = 'child scope';
    // 同在DOM中一样，我们可以通过当前$scope直接访问原型中的任意属性
    $scope.fullSentenceFromChild = 'Same $scope: We can access: ' +
        $scope.rootProperty + ' and ' +
        $scope.parentProperty + ' and ' +
        $scope.childProperty
});
```

在线示例：http://jsbin.com/URuyoG/1/edit，为方便学习，有些部分使用了彩色。

更多关于ng-controller的内容请查看9.2节。

注意，还有其他内置指令（比如ng-include和ng-view）也会创建新的子作用域，这意味着它们在被调用时行为和ng-controller类似。我们在构造自定义指令时也可以创建新的子作用域。

8.2　向指令中传递数据

回顾一下如何定义指令：

```
angular.module('myApp', [])
.directive('myDirective', function() {
    return {
        restrict: 'A',
        replace: true,
        template: '<a href="http://google.com">Click me to go to Google</a>'
    }
});
```

注意，我们在模板中硬编码了URL和链接文本：

```
template:'<a href="http://google.com"> Click me to go to Google</a>'
```

AngularJS并没有限制在指令的模板中硬编码字符串。

如果不将URL和链接文本混在指令内部，可以为其他使用我们指令的人提供更好的体验。我们的目标是关注指令的公共接口，就像其他任何编程语言一样。实际上，应该将上面的模板转换成可以接受两个变量的形式：一个变量是URL，另一个是链接文本：

```
template: '<a href="{{ myUrl }}">{{ myLinkText }}</a>'
```

在主HTML文档中，可以给指令添加myUrl和myLinkText两个属性，这两个参数会成为指令内部作用域的属性：

```
<div my-directive
    my-url="http://google.com"
    my-link-text="Click me to go to Google">
</div>
```

重新加载页面，注意声明指令的部分已经被模板代替，但是链接的href属性是空的，并且尖括号内也没有文本，如图8-6所示。

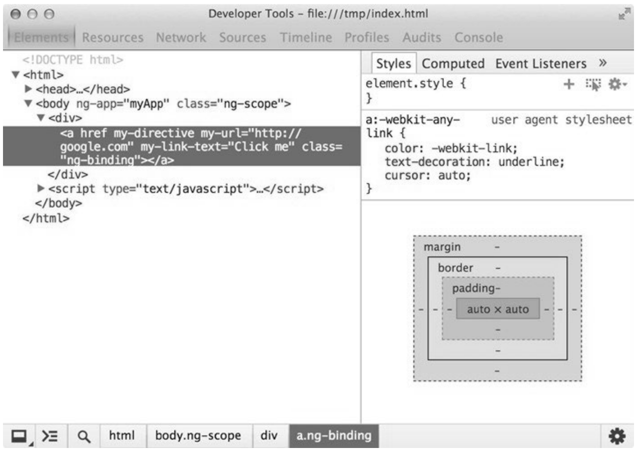


图8-6　更新模板

有好几种途径可以设置指令内部作用域中属性的值。最简单的方法就是使用由所属控制器提供的已经存在的作用域。

尽管简单，共享状态会导致很多其他问题。如果控制器被移除，或者在控制器的作用域中也定义了一个叫myUrl的属性，我们就被迫要修改代码，这是成本很高且让人沮丧的事情。

AngularJS允许通过创建新的子作用域或者隔离作用域来解决这个常见问题。

> 同之前在当前作用域介绍中介绍的继承作用域（子作用域）不同，隔离作用域同当前DOM的作用域是完全分隔开的。为了给这个新的对象设置属性，我们需要显式地通过属性传递数据，同在JavaScript或Ruby中给方法传递参数类似。

当用如下代码将指令的作用域设置成一个只包含它自己的属性的干净对象时：

```
scope: {
    someProperty: "needs to be set"
}
```

实际上创造的是隔离作用域。本质上，意味着指令有了一个属于自己的$scope对象，这个对

象只能在指令的方法中或指令的模板字符串中使用：

```
template: '<div>\
    we have access to {{ someProperty }}\
    </div>',
controller: function($scope) {
    //指令可以有它自己的控制器,
    // 那种情况下我们可以
    // => 错误!!!
    $scope.someProperty === "needs to be set";
}
```

错误？

目前为止，我们一直忽略了一个细节。实际上不能像上面的例子那样，在作用域对象内部直接设置someProperty属性。

```
scope: {
    // 这样行不通
    someProperty: 'needs to be set'
}
```

实际上要在DOM中像之前提到过的那样，像给函数传递参数一样，通过属性来设置值：

```
<div my-directive
    some-property="someProperty with @ binding">
</div>
```

现在，我们在作用域对象内部把someProperty值设置为@这个绑定策略。这个绑定策略告诉AngularJS将DOM中some-property属性的值复制给新作用域对象中的someProperty属性：

```
scope: {
    someProperty: '@'
}
```

注意，默认情况下someProperty在DOM中的映射是some-property属性。如果我们想显式指定绑定的属性名，可以用如下方式：

```
scope: {
    someProperty: '@someAttr'
}
```

在这个例子中，被绑定的属性名是some-attr而不是some-property。

```
<div my-directive
    some-attr="someProperty with @ binding">
</div>
```

现在，当我们在指令模板或控制器中（之前的例子这样做过）访问someProperty时，会得到DOM属性中的值的副本：

```
template:'<div>\
we have access to {{ someProperty }}\
    </div>',
    controller: function($scope) {
    // 指令可以有它自己的控制器，在那种情况下，我们可以将
    // $scope.someProperty设置成"someProperty with @ binding"
}
```

回到主题，我们用属性将数据从DOM中复制到指令的隔离作用域中：

```
<div my-directive
    my-url="http://google.com"
    my-link-text="Click me to go to Google"></div>

angular.module('myApp', [])
.directive('myDirective', function() {
    return {
        restrict: 'A',
        replace: true,
        scope: {
            myUrl: '@', //绑定策略
            myLinkText: '@' //绑定策略
        },
        template: '<a href="{{myUrl}}">' +
            '{{myLinkText}}</a>'
    };
});
```

在线示例：http://jsbin.com/eloKoDI/1/edit。

默认情况下约定DOM属性和JavaScript中对象属性的名字是一样的（除非对象的属性名采用的是驼峰式写法）。

由于作用域中属性经常是私有的，因此可以（虽然不常见）指定我们希望将这个内部属性同哪个DOM属性进行绑定：

```
scope: {
    myUrl: '@someAttr',
    myLinkText: '@'
}
```

上面的隔离作用域中的内容是：将指令的私有属性`$scope.myUrl`同DOM中`some-attr`属性的值绑定起来。这个值既可以是硬编码的也可以是当前作用域（例如`Some-attr="{{expression}}"`）中某个表达式的运算结果。

在DOM中要用`some-attr`代替`my-url`：

```
<div my-directive
     some-attr="http://google.com"
     my-link-text="Click me to go to Google" >
</div>
```

更进一步，还可以在DOM对应的作用域上运算表达式，并将结果传递给指令，在指令内部最终被绑定在属性上：

```
<div my-directive
     some-attr="{{ 'http://' + 'google.com' }}">
</div>
```

在此之上，我们来看看如何创建一个文本输入域，并将输入值同指令内部隔离作用域的属性绑定起来：

> 注意在输入标签上使用了内置指令ng-model。这个指令可以将输入文本同$scope上的myUrl属性进行绑定。

```
<input type="text" ng-model="myUrl" />
<div my-directive
    some-attr="{{ myUrl }}"
    my-link-text="Click me to go to Google">
    </div>
```

这段代码是可以工作的，但如果我们将文本输入字段移到指令内部并在另一个指令中进行绑定，就无法正常工作了：

```
<div my-directivesome-attr="{{ myUrl }}"
my-link-text="Click me to go to Google">
</div>
```

还有下面这段代码：

```
template: '<div>\
    <input type="text" ng-model="myUrl" />\
    <a href="{{myUrl}}">{{myLinkText}}</a>\
    </div>'
```

通过观察Chrome开发者工具中的href可以知道，我们并没有错误地将内部$scope的属性myUrl同外部的DOM属性some-attr绑定在一起。值是通过对DOM属性进行复制被传递到隔离作用域中的，难道它不应该同时设置同名属性的值吗？如图8-7所示。



图8-7　Chrome开发者工具

出现这种现象的原因是，内置指令ng-model在它自身内部的隔离作用域和DOM的作用域（由控制器提供）之间创建了一个双向数据绑定。

让我们来模仿一下这个设置过程以使例子能正工作。我们的目标是理解双向数据绑定，以及ng-model在这个过程中的行为。

双向数据绑定或许是AngularJS中最重要且无法通过jQuery简单实现的功能之一。我们需要自己实现它进而了解它的神奇效果，幸好，所需工作并不多。接下来在我们的隔离作用域和ng-model内部的隔离作用域之间创建一个双向数据绑定，这样我们的例子就完整了。将内部的$scope.myUrl属性同当前控制器作用域中的theirUrl属性进行绑定，在DOM中通过作用域查询来实现这个绑定。

在这个流程中，给两个方向的绑定都添加一个文本输入字段。通过这两个输入字段可以方便地观察作用域是如何在DOM中通过原型继承链接在一起的①：

```
<label>Their URL field:</label>
<input type="text"  ng-model="theirUrl">
<div my-directive
     some-attr="theirUrl"
     my-link-text="Click me to go to Google"></div>
angular.module('myApp', [])
.directive('myDirective', function() {
    return {
        restrict: 'A',
        replace: true,
        scope: {
            myUrl: '=someAttr', // 经过了修改
            myLinkText: '@'
        },
        template: '\
        <div>\
            <label>My Url Field:</label>\
            <input type="text"\
                ng-model="myUrl" />\
            <a href="{{myUrl}}">{{myLinkText}}</a>\
        </div>\
    };
});
```


在线示例：http://jsbin.com/IteNita/1/edit。

在Chrome开发者工具中一边在两个文本输入字段中输入内容，一边审查`href`属性的值，如图8-8所示，酷。



图8-8　Chrome开发者工具

除了将原来的文本输入字段添加回主HTML文档外，唯一的修改是用=绑定策略代替了`@`。

总地来说，这个例子展示了双向数据绑定的神奇效果，它是AngularJS的主要卖点之一。

了解内部指令的工作原理非常重要，这样才能在同自定义指令一起使用时把它们的行为考虑在内。

下一章我们会详细介绍AngularJS的内置指令，以便了解如何使用它们以及它们存在的意义。

了解完内置指令，第10章将详细介绍创建自定义指令时的高级设置。

最后，我们会创建自定义指令，然后讨论应用架构，在这个话题中自定义和内置指令都非常重要。

① 这个描述不是很准确，原型继承的机制应该是在AngularJS的JavaScript代码中而非DOM中实现的。——译者注

第 9 章

内置指令

AngularJS提供了一系列内置指令。其中一些指令重载了原生的HTML元素，比如`<form>`和`<a>`标签，当在HTML中使用标签时，并不一定能明确看出是否在使用指令。

例如，`<form>`标签被从底层扩展了一系列高级功能，包括表单验证等，原生HTML表单并不会提供这些功能。

其他内置指令通常以ng为前缀，很容易识别。例如后面将介绍的ng-href指令，它会提供一个超链接，这个链接将处于禁用状态，直到ng-href="someExpresiion"中的表达式被调用并且返回一个值。

最后，某些内置指令并不会有对应的HTML标签，比如ng-controller，这个指令可以在标签的属性中使用，通常在包含很多子元素并且需要共享作用域时使用。

注意，所有以ng前缀开头作为命名空间的指令都是AngularJS提供的内置指令，因此不要把你自己开发的指令以这个前缀命名。

9.1 基础 ng 属性指令

首先来看看和原生HTML标签名称相似的一组内置指令，这组指令非常容易记忆，因为仅仅是在原生标签名前加上了ng前缀，包括：

- ng-href；
- ng-src；
- ng-disabled；
- ng-checked；
- ng-readonly；
- ng-selected；
- ng-class；
- ng-style。

9.1.1 布尔属性

下面介绍的指令将帮助我们更简便地使用HTML布尔属性。

根据HTML标准的定义[①]，布尔属性代表一个true或false值。当这个属性出现时，这个属性的值就是true（无论实际定义的值是什么）。如果未出现，这个属性的值就是false。

当在AngularJS中使用动态数据绑定时，不能简单地将这个属性值设置为ture或false，因为根据标准定义只有当这个属性不出现时，它的值才为false。因此AngularJS提供了一组带有ng-前缀版本的布尔属性，通过运算表达式的值来决定在目标元素上是插入还是移除对应的属性。

1. ng-disabled

使用ng-disabled可以把disabled属性绑定到以下表单输入字段上：

- <input> （text、checkbox、radio、number、url，email、submit）；
- <textarea>；
- <select>；
- <button>。

当写普通的HTML输入字段时，如果在元素标签上出现了disabled属性就会禁用这个输入字段。通过ng-disabled可以对是否出现属性进行绑定。例如，在下面的例子中按钮会一直禁用，直到用户在文本字段中输入内容：

```
<input type="text" ng-model="someProperty" placeholder="TypetoEnable">
<button ng-model="button" ng-disabled="!someProperty">AButton</button>
```

在下面的例子，文本字段会被禁用五秒，直到在$timeout中将isDisabled属性设置为true：

```
<textarea ng-disabled="isDisabled">Wait5seconds</textarea>

angular.module('myApp', [])
.run(function($rootScope, $timeout) {
    $rootScope.isDisabled = true;
    $timeout(function() {
        $rootScope.isDisabled = false;
    }, 5000);
});
```

两个例子的在线示例：http://jsbin.com/iHiYItu/1/edit。

2. ng-readonly

同其他布尔属性一样，HTML定义只会检查readonly属性是否出现，而不是它的实际值。

通过ng-readonly可以将某个返回真或假的表达式同是否出现readonly属性进行绑定：

```
Type here to make sibling readonly:
<input type="text" ng-model="someProperty"><br/>
<input type="text"
    ng-readonly="someProperty"
    value="Some text here"/>
```

3. ng-checked

标准的checked属性是一个布尔属性，不需要进行赋值。通过ng-checked将某个表达式同是否出现checked属性进行绑定。

① http://www.w3.org/html/wg/drafts/html/master/infrastructure.html#boolean-attribute

在下面的例子中，我们通过ng-init指令将someProperty的值设置为true。将some Property同ng-checked绑定在一起，AngularJS会输出标准的HTML checked属性，这样默认会把复选框勾选：

```
<label>someProperty = {{someProperty}}</label>
<input type="checkbox"
       ng-checked="someProperty"
       ng-init="someProperty = true"
       ng-model="someProperty">
```

下面的例子刚好相反：

```
<label>someProperty = {{anotherProperty}}</label>
<input type="checkbox"
       ng-checked="anotherProperty"
       ng-init="anotherProperty = false"
       ng-model="anotherProperty">
```

注意，为了展示，这里用ng-model把someProperty和anotherProperty的值绑定到了对应的<label>标签里。

4. ng-selected

ng-selected可以对是否出现option标签的selected属性进行绑定：

```
<label>Select Two Fish:</label>
<input type="checkbox"
       ng-model="isTwoFish"><br/>
<select>
  <option>One Fish</option>
  <option ng-selected="isTwoFish">Two Fish</option>
</select>
```

在线示例：http://jsbin.com/oQazOQE/2/edit。

9.1.2　类布尔属性

ng-href、ng-src等属性虽然不是标准的HTML布尔属性，但是由于行为相似，所以在AngularJS源码内部是和布尔属性同等对待的，下面介绍这些属性。

ng-href和ng-src都能有效帮助重构和避免错误，因此在改进代码时强烈建议用它们代替原来的href和src属性。

1. ng-href

当使用当前作用域中的属性动态创建URL时，应该用ng-href代替href。

这里的潜在问题是当用户点击一个由插值动态生成的链接时，如果插值尚未生效，将会跳转到错误的页面（通常是404）。

这时，如果使用ng-href，AngularJS会等到插值生效（在例子中是两秒以后）后再执行点击链接的行为：

```
<!-- 当 href 包含一个 {{expression}}时总是使用 ng-href -->
<a ng-href="{{ myHref }}">I'm feeling lucky, when I load</a>
```

```
<!-- 用户单击之前，href不会加载 -->
<a href="{{ myHref }}">I'm feeling 404</a>
```

将插值生效的事件延迟两秒，来观察实际的行为：

```
angular.module('myApp', [])
.run(function($rootScope, $timeout) {
    $timeout(function() {
        $rootScope.myHref = 'http://google.com';
    }, 2000);
});
```

在线示例：http://jsbin.com/IgInopi/1/edit。

2. ng-src

AngularJS会告诉浏览器在ng-src对应的表达式生效之前不要加载图像：

```
<h1>WrongWay</h1>
<img src="{{imgSrc}}"/>

<h1>Rightway</h1>
<img ng-src="{{imgSrc}}"/>

angular.module('myApp', [])
.run(function($rootScope, $timeout) {
    $timeout(function() {
        $rootScope.imgSrc = 'https://www.google.com/images/srpr/logo11w.png';
    }, 2000);
});
```

在线示例：http://jsbin.com/egucIqU/1/edit。

浏览在线示例时，通过Chrome开发者工具的网络面板观察资源加载状况，注意，其中一个请求是红色的，说明发生了错误。这个错误是由于Wrong Way中我们用src代替了ng-src导致的。

9

9.2 在指令中使用子作用域

下面将要介绍的指令会以父级作用域为原型生成子作用域。这种继承的机制可以创建一个隔离层，用来将需要协同工作的方法和数据模型对象放置在一起。

ng-app和ng-controller是特殊的指令，因为它们会修改嵌套在它们内部的指令的作用域。

ng-app为AngularJS应用创建$rootScope，ng-controller则会以$rootScope或另外一个ng-controller的作用域为原型创建新的子作用域。

1. ng-app

任何具有ng-app属性的DOM元素将被标记为$rootScope的起始点。

$rootScope是作用域链的起始点，任何嵌套在ng-app内的指令都会继承它。

在JavaScript代码中通过run方法来访问$rootScope。

```
<html ng-app="myApp">
    <body>
        {{ someProperty }}
```

```
        <button ng-click="someAction()"></button>
    </body>
</html>

angular.module('myApp', [])
.run(function($rootScope) {
    $rootScope.someProperty = 'hello computer';
    $rootScope.someAction = function() {
        $rootScope.someProperty = 'hello human';
    };
});
```

在线示例：http://jsbin.com/ICOzeFI/2/edit。

这里为了演示方便，像使用全局作用域一样使用$rootScope，实际生产中不建议这样做。

可以在整个文档中只使用一次ng-app。如果需要在一个页面中放置多个AngularJS应用，需要手动引导应用。第24章会深入讨论手动引导应用。

2. ng-controller

内置指令ng-controller的作用是为嵌套在其中的指令创建一个子作用域，避免将所有操作和模型都定义在$rootScope上。用这个指令可以在一个DOM元素上放置控制器。

ng-controller接受一个参数expression，这个参数是必需的。

expression参数是一个AngularJS表达式。

子$scope只是一个JavaScript对象，其中含有从父级$scope中通过原型继承得到的方法和属性，包括应用的$rootScope。

嵌套在ng-controller中的指令有访问新子$scope的权限，但是要牢记每个指令都应该遵守的和作用域相关的规则。

回想一下，$scope对象的职责是承载DOM中指令所共享的操作和模型。

操作指的是$scope上的标准JavaScript方法。

模型指的是$scope上保存的包含瞬时状态数据的JavaScript对象。持久化状态的数据应该保存到服务中，服务的作用是处理模型的持久化。

出于技术和架构方面的原因，绝对不要直接将控制器中的$scope赋值为值类型对象（字符串、布尔值或数字）。DOM中应该始终通过点操作符.来访问数据。遵守这个规则将使你远离不可预期的麻烦。

控制器应该尽可能简单。虽然可以用控制器来组织所有功能，但是将业务逻辑移到服务和指令中是非常好的主意。查看第21章中的详细内容。

有了控制器，我们可以将之前的例子改造一下，把数据和操作放到子作用域中：

```
<div ng-controller="SomeController">
```

```
    {{ someModel.someProperty }}
    <button ng-click="someAction()">Communicate</button>
</div>

angular.module('myApp', [])
.controller('SomeController', function($scope) {
    //创建模型
    $scope.someModel = {
        // 添加属性
        someProperty: 'hello computer'
    }
    // 设置$scope自身的操作
    $scope.someAction = function() {
        $scope.someModel.someProperty = 'hello human';
    };
});
```

在线示例：http://jsbin.com/OYikipe/1/edit。

注意，这个例子和之前相比有两处不同：首先，我们使用了`$rootScope`的子作用域，它提供了一个干净的对象供我们操作。使用子作用域意味着其上的数据模型和操作在应用的其他地方是无法访问的，只能被这个作用域内的指令及其子作用域访问。其次，显式声明了数据模型，我们说过，这非常重要。为了展示这为什么重要，看一下这个例子的变体。这个例子中，在已有的控制器中嵌套了第二个控制器，并且没有设置模型对象的属性：

```
<div ng-controller="SomeController">
    {{ someBareValue }}
    <button ng-click="someAction()">Communicate to child</button>
    <div ng-controller="ChildController">
        {{ someBareValue }}
        <button ng-click="childAction()">Communicate to parent</button>
    </div>
</div>

angular.module('myApp', [])
.controller('SomeController', function($scope) {
    // 反模式，裸值
    $scope.someBareValue = 'hello computer';
    // 设置 $scope 本身的操作，这样没问题
    $scope.someAction = function() {
        // 在SomeController和ChildController中设置{{ someBareValue }}
        $scope.someBareValue = 'hello human, from parent';
    };
})
.controller('ChildController', function($scope) {
    $scope.childAction = function() {
        // 在ChildController中设置{{ someBareValue }}
        $scope.someBareValue = 'hello human, from child';
    };
});
```

在线示例：http://jsbin.com/UbIRIHa/1/。

由于原型继承的关系，修改父级对象中的`someBareValue`会同时修改子对象中的值，但反之则不行。

可以看下这个例子的实际效果，首先点击child button，然后点击parent button。这个例子充分说明了子控制器是复制而非引用`someBareValue`。

9

> JavaScript对象要么是值复制要么是引用复制。字符串、数字和布尔型变量是值复制。数组、对象和函数则是引用复制。

如果将模型对象的某个属性设置为字符串，它会通过引用进行共享，因此在子$scope中修改属性也会修改父$scope中的这个属性。下面的例子展示了正确的做法：

```
<div ng-controller="SomeController">
    {{ someModel.someValue }}
    <button ng-click="someAction()">Communicate to child</button>
    <div ng-controller="ChildController">
    {{ someModel.someValue }}
        <button ng-click="childAction()">Communicate to parent</button>
    </div>
</div>

angular.module('myApp', [])
.controller('SomeController', function($scope) {
    // 最佳实践，永远使用一个模式
    $scope.someModel = {
        someValue: 'hello computer'
    }
    $scope.someAction = function() {
        $scope.someModel.someValue = 'hello human, from parent';
    };
})
.controller('ChildController', function($scope) {
    $scope.childAction = function() {
        $scope.someModel.someValue = 'hello human, from child';
    };
});
```

在线示例：http://jsbin.com/afIyeda/1/edit。

无论点击哪个按钮，值都会进行同步修改。

注意，虽然这个特性是使用ng-controller时最重要的特性之一，但在使用任何会创建子作用域的指令时，如果将指令定义中的scope设置为true，这个特性也会带来负面影响。下面的内置指令都有同样的特性：

- ng-include
- ng-switch
- ng-repeat
- ng-view
- ng-controller
- ng-if

3. ng-include

使用ng-include可以加载、编译并包含外部HTML片段到当前的应用中。模板的URL被限制在与应用文档相同的域和协议下，可以通过白名单或包装成被信任的值来突破限制。更进一步，需要考虑跨域资源共享（Cross-Origin Resource Sharing，CORS）和同源规则（Same Origin Policy）来确保模板可以在任何浏览器中正常加载。例如，所有浏览器都不能进行跨域的请求，部分浏览器也不能访问file://协议的资源。

在开发中，可以通过命令行命令chrome --allow-file-access-from-files来禁止CORS错误。只在紧急情况下使用这个方法，比如你的老板正站在你身后，并且所有事情都无法正常工作。

在同一个元素上添加onload属性可以在模板加载完成后执行一个表达式。

要记住，使用ng-include时AngularJS会自动创建一个子作用域。如果你想使用某个特定的作用域，例如ControllerA的作用域，必须在同一个DOM元素上添加ng-controller ="ControllerA"指令，这样当模板加载完成后，不会像往常一样从外部作用域继承并创建一个新的子作用域。下面看一个例子：

```
<div ng-include="/myTemplateName.html"
     ng-controller="MyController"
     ng-init="name = 'World'">
    Hello {{ name }}
</div>
```

4. ng-switch

这个指令和ng-switch-when及on="propertyName"一起使用，可以在propertyName发生变化时渲染不同指令到视图中。在下面的例子中，当person.name是Ari时，文本域下面的div会显示出来，并且这个人会获得胜利：

```
<input type="text" ng-model="person.name"/>
<div ng-switch on="person.name">
    <p ng-switch-default>And the winner is</p>
    <h1 ng-switch-when="Ari">{{ person.name }}</h1>
</div>
```


注意，在switch被调用之前我们用ng-switch-default来输出默认值。

在线示例：http://jsbin.com/AVihUdi/2/。

5. ng-view

ng-view指令用来设置将被路由管理和放置在HTML中的视图的位置。第12章会深入研究这些内容。

查看第12章获得更详细信息。

6. ng-if

使用ng-if指令可以完全根据表达式的值在DOM中生成或移除一个元素。如果赋值给ng-if的表达式的值是false，那对应的元素将会从DOM中移除，否则对应元素的一个克隆将被重新插入DOM中。

ng-if同no-show和ng-hide指令最本质的区别是，它不是通过CSS显示或隐藏DOM节点，而是真正生成或移除节点。

当一个元素被ng-if从DOM中移除，同它关联的作用域也会被销毁。而且当它重新加入DOM中时，会通过原型继承从它的父作用域生成一个新的作用域。

同时有一个重要的细节需要知道，ngIf重新创建元素时用的是它们编译后的状态。如果ng-if

内部的代码加载之后被jQuery修改过（例如用.addClass），那么当ng-if的表达式值为false时，这个DOM元素会被移除，表达式再次成为true时这个元素及其内部的子元素会被重新插入DOM，此时这些元素的状态会是它们的原始状态，而不是它们上次被移除时的状态。也就是说无论用jQuery的.addClass添加了什么类都不会存在了。

```
<div ng-if="2+2===5">
    Won't see this DOM node, not even in the source code
</div>

<div ng-if="2+2===4">
    Hi, I do exist
</div>
```

在线示例：http://jsbin.com/ezEcamo/1/。

7. ng-repeat

ng-repeat用来遍历一个集合或为集合中的每个元素生成一个模板实例。集合中的每个元素都会被赋予自己的模板和作用域。同时每个模板实例的作用域中都会暴露一些特殊的属性。

- $index：遍历的进度（0...length-1）。
- $first：当元素是遍历的第一个时值为true。
- $middle：当元素处于第一个和最后元素之间时值为true。
- $last：当元素是遍历的最后一个时值为true。
- $even：当$index值是偶数时值为true。
- $odd：当$index值是奇数时值为true。

下面的例子展示了如何用$odd和$even来制作一个红蓝相间的列表。记住，JavaScript中数组的索引从0开始，因此我们用!$even和!$odd来将$even和$odd的布尔值反转。

```
<ul ng-controller="PeopleController">
    <li ng-repeat="person in people" ng-class="{even: !$even, odd: !$odd}">
        {{person.name}} lives in {{person.city}}
    </li>
</ul>

.odd {
    background-color: blue;
}
.even {
    background-color: red;
}

angular.module('myApp',[])
.controller('PeopleController',function($scope) {
    $scope.people = [
        {name: "Ari", city: "San Francisco"},
        {name: "Erik", city: "Seattle"}
    ];
});
```

在线示例：http://jsbin.com/akuYUkey/1/edit。

8. ng-init

ng-init指令用来在指令被调用时设置内部作用域的初始状态。

ng-init最常见的使用场景是：在类似本节的例子中那样，需要创建小的示例代码的时候。

对于任何需要健壮结构的场景，请在控制器中用数据模型对象来设置状态。

```
<div ng-init="greeting='Hello';person='World'">
    {{greeting}} {{person}}
</div>
```

在线示例：http://jsbin.com/OZENuhO/1/。

9. {{ }}

```
<div>{{name}}</div>
```

{{ }}语法是AngularJS内置的模板语法，它会在内部$scope和视图之间创建绑定。基于这个绑定，只要$scope发生变化，视图就会随之自动更新。

事实上它也是指令，虽然看起来并不像，实际上它是ng-bind的简略形式，用这种形式不需要创建新的元素，因此它常被用在行内文本中。

注意，在屏幕可视的区域内使用{{ }}会导致页面加载时未渲染的元素发生闪烁，用ng-bind可以避免这个问题。

```
<body ng-init="greeting='HelloWorld'">
    {{ greeting }}
</body>
```

在线示例：http://jsbin.com/ODUxeho/1/edit。

10. ng-bind

尽管可以在视图中使用{{ }}模板语法（AngularJS内置的方式），我们也可以通过ng-bind指令实现同样的行为。

```
<body ng-init="greeting='HelloWorld'">
    <p ng-bind="greeting"></p>
</body>
```

在线示例：http://jsbin.com/esihUJ/1/edit。

HTML加载含有{{ }}语法的元素后并不会立刻渲染它们，导致未渲染内容闪烁（Flash of Unrendered Content，FOUC）。我可以用ng-bind将内容同元素绑定在一起避免FOUC。内容会被当作子文本节点渲染到含有ng-bind指令的元素内。

11. ng-cloak

除使用ng-bind来避免未渲染元素闪烁，还可以在含有{{ }}的元素上使用ng-cloak指令：

```
<body ng-init="greeting='HelloWorld'">
    <p ng-cloak>{{ greeting }}</p>
</body>
```

ng-cloak指令会将内部元素隐藏，直到路由调用对应的页面时才显示出来。

在线示例：http://jsbin.com/AJEboLO/1/edit。

12. ng-bind-template

同ng-bind指令类似，ng-bind-template用来在视图中绑定多个表达式。

```
<div
ng-bind-template="{{message}}{{name}}">
</div>
```

13. ng-model

ng-model指令用来将input、select、text area或自定义表单控件同包含它们的作用域中的属性进行绑定。它可以提供并处理表单验证功能，在元素上设置相关的CSS类（ng-valid、ng-invalid等），并负责在父表单中注册控件。

它将当前作用域中运算表达式的值同给定的元素进行绑定。如果属性并不存在，它会隐式创建并将其添加到当前作用域中。

我们应该始终用ngModel来绑定$scope上一个数据模型内的属性，而不是$scope上的属性，这可以避免在作用域或后代作用域中发生属性覆盖。

例如：

```
<input type="text"
    ng-model="modelName.someProperty" />
```

上面的例子展示了如何正确地考虑和使用ngModel。

14. ng-show/ng-hide

ng-show和ng-hide根据所给表达式的值来显示或隐藏HTML元素。当赋值给ng-show指令的值为false时元素会被隐藏。类似地，当赋值给ng-hide指令的值为true时元素也会被隐藏。

元素的显示或隐藏是通过移除或添加ng-hide这个CSS类来实现的。.ng-hide类被预先定义在了AngularJS的CSS文件中，并且它的display属性的值为none（用了!important标记）。

```
<div ng-show="2 + 2 == 5">
    2 + 2 isn't 5, don't show
</div>
<div ng-show="2 + 2 == 4">
    2 + 2 is 4, do show
</div>
<div ng-hide="2 + 2 == 5">
    2 + 2 isn't 5, don't hide
</div>

<div ng-hide="2 + 2 == 4">
    2 + 2 is 4, do hide
</div>
```

在线示例：http://jsbin.com/ihOkagE/1/。

15. ng-change

这个指令会在表单输入发生变化时计算给定表达式的值。因为要处理表单输入，这个指令要和ngModel联合起来使用。

```
<div ng-controller="EquationController">
    <input type="text"
```

```
        ng-model="equation.x"
        ng-change="change()" />
    <code>{{ equation.output }}</code>
</div>


angular.module('myApp',[])
.controller('EquationController',function($scope) {
    $scope.equation = {};
    $scope.change = function() {
        $scope.equation.output
            = parseInt($scope.equation.x) + 2;
    };
});
```

在线示例：http://jsbin.com/onUXuxO/1/edit。

上面的例子中，只要文本输入字段中的内容发生了变化就会改变equation.x的值，进而运行change()函数。

16. **ng-form**

ng-form用来在一个表单内部嵌套另一个表单。普通的HTML <form>标签不允许嵌套，但ng-form可以。

这意味着内部所有的子表单都合法时，外部的表单才会合法。这对于用ng-repeat动态创建表单是非常有用的。

由于不能通过字符插值来给输入元素动态地生成name属性，所以需要将ng-form指令内每组重复的输入字段都包含在一个外部表单元素内。

下面的CSS类会根据表单的验证状态自动设置：

- 表单合法时设置ng-valid；
- 表单不合法时设置ng-invlid；
- 表单未进行修改时设置ng-pristion；
- 表单进行过修改时设置ng-dirty。

Angular不会将表单提交到服务器，除非它指定了action属性。要指定提交表单时调用哪个JavaScript方法，使用下面两个指令中的一个。

- ng-submit：在表单元素上使用。
- ng-click：在第一个按钮或submit类型（input[type=submit]）的输入字段上使用。

为了避免处理程序被多次调用，只使用下面两个指令中的一个。

下面的例子展示了如何通过服务器返回的JSON数据动态生成一个表单。我们用ng-loop来遍历从服务器取回的所有数据。由于不能动态生成name属性，而我们又需要这个属性做验证，所以在循环的过程中会为每一个字段都生成一个新表单。

由于AngularJS中用来取代<form>的ng-form指令可以嵌套，并且外部表单在所有子表单都合法之前一直处于不合法状态，因此我们可以在动态生成子表单的同时使用表单验证功能。是的，鱼和熊掌可以兼得。

下面先看一下我们硬编码的JSON数据，把它假设成是从服务器返回的：

9

```
angular.module('myApp',[])
.controller('FormController',function($scope) {
    $scope.fields = [
        {placeholder: 'Username', isRequired: true},
        {placeholder: 'Password', isRequired: true},
        {placeholder: 'Email (optional)', isRequired: false}
    ];

    $scope.submitForm = function() {
        alert("it works!");
    };
});
```

下面用这些数据生成一个有验证功能的动态表单：

```
<form name="signup_form"
    ng-controller="FormController"
    ng-submit="submitForm()" novalidate>
    <div ng-repeat="field in fields"
            ng-form="signup_form_input">
        <input type="text"
            name="dynamic_input"
            ng-required="field.isRequired"
            ng-model="field.name"
            placeholder="{{field.placeholder}}" />
      <div
            ng-show="signup_form_input.dynamic_input.$dirty &&
            signup_form_input.dynamic_input.$invalid">
            <span class="error"
                ng-show="signup_form_input.dynamic_input.$error.required">
                                    The field is required.
            </span>
        </div>
    </div>
    <button type="submit"
        ng-disabled="signup_form.$invalid">
           Submit All
    </button>
</form>
input.ng-invalid {
    border: 1px solid red;
}


input.ng-valid {
    border: 1px solid green;
}
```

在线示例：http://jsbin.com/UduNeCA/1/edit。

17. ng-click

ng-click用来指定一个元素被点击时调用的方法或表达式。

```
<div ng-controller="CounterController">
    <button ng-click="count = count + 1"
            ng-init="count=0">
        Increment
    </button>
    count: {{ count }}
    <button ng-click="decrement()">
```

```
        Decrement
    </button>
<div>

angular.module('myApp',[])
.controller('CounterController', function($scope) {
    $scope.decrement = function() {
        $scope.count = $scope.count - 1;
    };
})
```

在线示例：http://jsbin.com/uGipUBU/2/edit。

18. **ng-select**

ng-select用来将数据同HTML的<select>元素进行绑定。这个指令可以和ng-model以及ng-options指令一同使用，构建精细且表现优良的动态表单。

ng-options的值可以是一个*内涵表达式*（comprehension expression），其实这只是一种有趣的说法，简单来说就是它可以接受一个数组或对象，并对它们进行循环，将内部的内容提供给select标签内部的选项。它可以是下面两种形式。

- 数组作为数据源：
  - 用数组中的值做标签；
  - 用数组中的值作为选中的标签；
  - 用数组中的值做标签组；
  - 用数组中的值作为选中的标签组。
- 对象作为数据源：
  - 用对象的键–值（key-value）做标签；
  - 用对象的键–值作为选中的标签；
  - 用对象的键–值作为标签组；
  - 用对象的键–值作为选中的标签组。

下面看一个ng-select指令的实例：

```
<div ng-controller="CityController">
    <select ng-model="city"
    ng-options="city.name for city in cities">
        <option value="">Choose City</option>
    </select>
    Best City: {{ city.name }}
</div>

angular.module('myApp',[])
.controller('CityController',function($scope) {
    $scope.cities = [
      {name: 'Seattle'},
      {name: 'San Francisco'},
      {name: 'Chicago'},
      {name: 'New York'},
      {name: 'Boston'}
    ];
});
```

在线示例：http://jsbin.com/iQelOxi/1/edit。

19. ng-submit

ng-submit用来将表达式同onsubmit事件进行绑定。这个指令同时会阻止默认行为（发送请求并重新加载页面），除非表单不含有action属性。

```
<form ng-submit="submit()"
      ng-controller="FormController">
    Enter text and hit enter:
    <input type="text"
          ng-model="person.name"
          name="person.name" />
    <input type="submit"
          name="person.name"
          value="Submit" />
    <code>people={{people}}</code>
    <ul ng-repeat="(index, object) in people">
        <li>{{ object.name }}</li>
    </ul>
</form>

angular.module('myApp',[])
.controller('FormController',function($scope) {

    $scope.person = {
        name: null
    };

    $scope.people = [];

    $scope.submit = function() {
        if ($scope.person.name) {
            $scope.people.push({name: $scope.person.name});
            $scope.person.name = '';
        }
    };
});
```

在线示例：http://jsbin.com/ONIcAC/1/edit。

20. ng-class

使用ng-class 动态设置元素的类，方法是绑定一个代表所有需要添加的类的表达式。重复的类不会添加。当表达式发生变化，先前添加的类会被移除，新类会被添加。

下面的例子会用ng-class在一个随机数大于5时将.red类添加到一个div上：

```
<div ng-controller="LotteryController">
    <div ng-class="{red: x > 5}"
        ng-if="x > 5">
        You won!
    </div>
    <button ng-click="x = generateNumber()"
        ng-init="x = 0">
        Draw Number
    </button>
    <p>Number is: {{ x }}</p>
</div>
```

```
.red {
    background-color: red;
}

angular.module('myApp',[])
.controller('LotteryController', function($scope) {
    $scope.generateNumber = function() {
        return Math.floor((Math.random()*10)+1);
    };
});
```

在线示例：http://jsbin.com/IvEcUci/1/edit。

21. ng-attr-(suffix)

当AngularJS编译DOM时会查找花括号{{ some expression }}内的表达式。这些表达式会被自动注册到$watch服务中并更新到$digest循环中，成为它的一部分：

```
<-- updated when`someExpression` on the$scope
    is updated -->
<h1>Hello{{someExpression}}</h1>
```

有时浏览器会对属性会进行很苛刻的限制。SVG就是一个例子：

```
<svg>
    <circle cx="{{ cx }}"></circle>
</svg>
```

运行上面的代码会抛出一个错误，指出我们有一个非法属性。可以用ng-attr-cx来解决这个问题。注意，cx位于这个名称的尾部。在这个属性中，通过用{{ }}来写表达式，达到前面提到的目的。

```
<svg>
    <circle ng-attr-cx="{{ cx }}"><circle>
</svg>
```

9

第 10 章 指令详解

本章的目标是全面介绍指令的设置选项和功能，以及如何使用指令来打造成熟的客户端应用。

10.1 指令定义

对于指令，可以把它简单的理解成在特定DOM元素上运行的函数，指令可以扩展这个元素的功能。

例如，ng-click可以让一个元素能够监听click事件，并在接收到事件的时候执行AngularJS表达式。正是指令使得AngularJS这个框架变得强大，并且正如所见，我们可以自己创造新的指令。

AngularJS应用的模块中有很多方法可以使用，其中directive()这个方法是用来定义指令的：

```
angular.module('myApp', [])
.directive('myDirective', function ($timeout, UserDefinedService) {
    // 指令定义放在这里
});
```

directive() 方法可以接受两个参数：

1. name（字符串）

指令的名字，用来在视图中引用特定的指令。

2. factory_function （函数）

这个函数返回一个对象，其中定义了指令的全部行为。$compile服务利用这个方法返回的对象，在DOM调用指令时来构造指令的行为。

```
angular.application('myApp', [])
.directive('myDirective', function() {
    // 一个指令定义对象
    return {
        // 通过设置项来定义指令，在这里进行覆写
    };
});
```

我们也可以返回一个函数代替对象来定义指令，但是像上面的例子一样，通过对象来定义是最佳的方式。当返回一个函数时，这个函数通常被称作链接传递（postLink）函数，利用它我们可以定义指令的链接（link）功能。由于返回函数而不是对象会限制定义指令时的自由度，因此只在构造简单的指令时才比较有用。

当AngularJS启动应用时，它会把第一个参数当作一个字符串，并以此字符串为名来注册第二个参数返回的对象。AngularJS编译器会解析主HTML的DOM中的元素、属性、注释和CSS类名中使用了这个名字的地方，并在这些地方引用对应的指令。当它找到某个已知的指令时，就会在页面中插入指令所对应的DOM元素。

```
<div my-directive></div>
```

为了避免与未来的HTML标准冲突，给自定义的指令加入前缀来代表自定义的命名空间。AngularJS本身已经使用了ng-前缀，所以可以选择除此以外的名字。在例子中我们使用my-前缀（比如my-derictive）。

指令的工厂函数只会在编译器第一次匹配到这个指令时调用一次。和controller函数类似，我们通过$injetor.invoke来调用指令的工厂函数。

当AngularJS在DOM中遇到具名的指令时，会去匹配已经注册过的指令，并通过名字在注册过的对象中查找。此时，就开始了一个指令的生命周期，指令的生命周期开始于$compile方法并结束于link方法，在本章后面的内容中我们会详细介绍这个过程。

下面，来看看定义一个指令时可以使用的全部设置选项。

一个JavaScript对象由键和值组成。当一个给定键的值被设置为一个字符串、布尔值、数字、数组或对象时，我们把这个键称为属性。当把键设置为函数时，我们把它叫做方法。

10

可能的选项如下所示，每个键的值说明了可以将这个属性设置为何种类型或者什么样的函数：

```
angular.module('myApp', [])
.directive('myDirective', function() {
    return {
        restrict: String,
        priority: Number,
        terminal: Boolean,
        template: String or Template Function:
            function(tElement, tAttrs) (...},
        templateUrl: String,
        replace: Boolean or String,
        scope: Boolean or Object,
        transclude: Boolean,
        controller: String or
        function(scope, element, attrs, transclude, otherInjectables) { ... },
        controllerAs: String,
        require: String,
        link: function(scope, iElement, iAttrs) { ... },
        compile: // 返回一个对象或连接函数，如下所示：
            function(tElement, tAttrs, transclude) {
                return {
                    pre: function(scope, iElement, iAttrs, controller) { ... },
                    post: function(scope, iElement, iAttrs, controller) { ... }
                }
                // 或者
                return function postLink(...) { ... }
```

```
        }
    };
});
```

10.1.1 restrict（字符串）

restrict是一个可选的参数。它告诉AngularJS这个指令在DOM中可以何种形式被声明。默认AngularJS认为restrict的值是A，即以属性的形式来进行声明。

可选值如下：

E（元素）

```
<my-directive></my-directive>
```

A（属性，默认值）

```
<div my-directive="expression"></div>
```

C（类名）

```
<div class="my-directive:expression;"></div>
```

M（注释）

```
<--directive:my-directive expression-->
```

这些选项可以单独使用，也可以混合在一起使用：

```
angular.module('myDirective', function(){
    return {
        restrict: 'EA' // 输入元素或属性
    };
});
```

上面的配置可以同时用属性或注释的方式来声明指令：

```
<-- 作为一个属性 -->
<div my-directive></div>
<-- 或者作为一个元素 -->
<my-directive></my-directive>
```

属性是用来声明指令最常用的方式，因为它能在包括老版本的IE浏览器在内的所有浏览器中正常工作，并且不需要在文档头部注册新的标签。更多内容请查看第30章。

> 尽量避免用注释方式来声明属性。这种方式最初被用来声明由多个标签组成的指令。这种方法在某些情况下特别有用，比如在<table>元素内使用ng-repeat指令，但在AngularJS 1.2中ng-repeat可以通过ng-repeat-start和ng-repeat-end来更优雅地满足这个需求，注释模式就没有什么用武之地了。如果你对此很好奇，可以通过Chrome开发者工具的element标签观察一下使用ng-repeat时被隐式添加的注释。

元素方式还是属性方式

在页面中通过元素方式创建新的指令可以将一些功能封装在元素内部。例如，如果我们想要

做一个时钟（忽略对老版本IE浏览器的支持），可以创建一个clock指令，然后在DOM中用如下代码来声明：

```
<my-clock></my-clock>
```

这样做可以告诉指令的使用者，这里会完整包含应用的某一部分内容。这个时钟并不是对一个既有时钟的修饰或扩展，而是一个全新的单元。尽管这里也可以使用属性形式声明指令（AngularJS并不在意），但我们选择了元素形式，因为这样可以更明确地表达意图。

用属性形式来给一个已经存在的元素添加数据或行为。以时钟为例，假设我们更喜欢模拟时钟：

```
<my-clock clock-display="analog"></my-clock>
```

如何进行选择，通常取决于定义的指令是否包含某个组件的核心行为，或者用额外的行为、状态或者其他内容（比如模拟时钟）对某个核心组件进行修饰或扩展。

使用何种指令声明格式的指导原则是能够准确表达每一段代码的意图，创造易于理解和分享的清晰代码。

另外一个重要的标准，是根据指令是否创建、继承或将自己从所属的环境中隔离出去进行判断。指令的父子关系对其组成和重用性起着至关重要的作用，会有额外的内容来更加深入地讨论指令的作用域。

10.1.2 优先级（数值型）

优先级参数可以被设置为一个数值。大多数指令会忽略这个参数，使用默认值0，但也有些场景设置高优先级是非常重要甚至是必须的。例如，ngRepeat将这个参数设置为1000，这样就可以保证在同一元素上，它总是在其他指令之前被调用。

如果一个元素上具有两个优先级相同的指令，声明在前面的那个会被优先调用。如果其中一个的优先级更高，则不管声明的顺序如何都会被优先调用：具有更高优先级的指令总是优先运行。

> ngRepeat是所有内置指令中优先级最高的，它总是在其他指令之前运行。这样设置主要考虑的是性能。在讨论编译参数时会更详细介绍性能相关的内容。

10.1.3 `terminal`（布尔型）

terminal是一个布尔型参数，可以被设置为true或false。

这个参数用来告诉AngularJS停止运行当前元素上比本指令优先级低的指令。但同当前指令优先级相同的指令还是会被执行。

如果元素上某个指令设置了terminal参数并具有较高的优先级，就不要再用其他低优先级的指令对其进行修饰了，因为不会被调用。但是具有相同优先级的指令还是会被继续调用。

使用了terminal参数的例子是ngView和ngIf。ngIf的优先级略高于ngView，如果ngIf的表达式值为true，ngView就可以被正常执行，但如果ngIf表达式的值为false，由于ngView的优先级较低就不会被执行。

10.1.4 template（字符串或函数）

template参数是可选的，必须被设置为以下两种形式之一：

- 一段HTML文本；
- 一个可以接受两个参数的函数，参数为tElement和tAttrs，并返回一个代表模板的字符串。tElement和tAttrs中的t代表template，是相对于instance的。在讨论链接和编译设置时会详细介绍，模板元素或属性与实例元素或属性之间的区别。

AngularJS会同处理HTML一样处理模板字符串。模板中可以通过大括号标记来访问作用域，例如{{ expression }}。

如果模板字符串中含有多个DOM元素，或者只由一个单独的文本节点构成，那它必须被包含在一个父元素内。换句话说，必须存在一个根DOM元素：

```
template: '\
    <div> <-- single root element -->\
        <a href="http://google.com">Click me</a>\
        <h1>When using two elements, wrap them in a parent element</h1>\
    </div>\
```

另外，注意每一行末尾的反斜线，这样AngularJS才能正确解析多行字符串。在实际生产中，更好的选择是使用templateUrl参数引用外部模板，因为多行文本阅读和维护起来都是一场噩梦。

模板字符串和templateURL中最需要了解的重要功能，是它们如何同作用域链接起来。第8章讨论了作用域是如何传递给指令的。

10.1.5 templateUrl（字符串或函数）

templateUrl是可选的参数，可以是以下类型：

- 一个代表外部HTML文件路径的字符串；
- 一个可以接受两个参数的函数，参数为tElement和tAttrs，并返回一个外部HTML文件路径的字符串。

无论哪种方式，模板的URL都将通过AngularJS内置的安全层，特别是$getTrustedResourceUrl，这样可以保护模板不会被不信任的源加载。

默认情况下，调用指令时会在后台通过Ajax来请求HTML模板文件。有两件事情需要知道。

- 在本地开发时，需要在后台运行一个本地服务器，用以从文件系统加载HTML模板，否则会导致Cross Origin Request Script（CORS）错误。
- 模板加载是异步的，意味着编译和链接要暂停，等待模板加载完成。

通过Ajax异步加载大量的模板将严重拖慢一个客户端应用的速度。为了避免延迟，可以在部署应用之前对HTML模板进行缓存。在大多数场景下缓存都是一个非常好的选择，因为AngularJS通过减少请求数量提升了性能。更多关于缓存的内容请查看第28章。

模板加载后，AngularJS会将它默认缓存到$templateCache服务中。在实际生产中，可以提前将模板缓存到一个定义模板的JavaScript文件中，这样就不需要通过XHR来加载模板了。更多

内容请查看第34章。

10.1.6 replace（布尔型）

replace是一个可选参数，如果设置了这个参数，值必须为true，因为默认值为false。默认值意味着模板会被当作子元素插入到调用此指令的元素内部：

```
<div some-directive></div>
.directive('someDirective', function() {
    return {
        template: '<div>some stuff here<div>'
    };
});
```

调用指令之后的结果如下（这是默认replace为false时的情况）：

```
<div some-directive>
    <div>some stuff here<div>
</div>
```

如果replace被设置为了true：

```
.directive('someDirective', function() {
    return {
        replace: true // 修饰过
        template: '<div>some stuff here<div>'
    };
});
```

指令调用后的结果将是：

```
<div>some stuff here<div>
```


10.2　指令作用域

为了完全理解指令定义对象中剩下的参数，需要先介绍指令作用域是如何工作的。

$rootScope这个特殊的对象会在DOM中声明ng-app时被创建：

```
<div ng-app="myApp"
    ng-init="someProperty = 'some data'"></div>
<div ng-init="siblingProperty = 'more data'">
        Inside Div Two
    <div ng-init="aThirdProperty"></div>
</div>
```

上面的代码中，我们在应用的根作用域中设置了三个属性：someProperty、siblingProperty和anotherSiblingProperty。

从这里开始，DOM中每个指令调用时都可能会：

- 直接调用相同的作用域对象；
- 从当前作用域对象继承一个新的作用域对象；
- 创建一个同当前作用域相隔离的作用域对象。

上面的列子展示的是第一种情况。前两个div是兄弟元素，可以通过get和set访问$rootScope。第二个div内部的div同样可以通过get和set访问相同的根作用域。

指令嵌套并不一定意味着需要改变它的作用域。默认情况下，子指令会被付予访问父DOM元素对应的作用域的能力，这样做的原因可以通过介绍指令的scope参数来理解，scope参数默认是false。

10.2.1　scope参数（布尔型或对象）

scope参数是可选的，可以被设置为true或一个对象。默认值是false。

当scope设置为true时，会从父作用域继承并创建一个新的作用域对象。

如果一个元素上有多个指令使用了隔离作用域，其中只有一个可以生效。只有指令模板中的根元素可以获得一个新的作用域。因此，对于这些对象来说scope默认被设置为true。

内置指令ng-controller的作用，就是从父级作用域继承并创建一个新的子作用域。它会创建一个新的从父作用域继承而来的子作用域。

用这些新内容更新一下前面的例子：

```
<div ng-app="myApp"
    ng-init="someProperty = 'some data'">
    <div ng-init="siblingProperty='moredata'">
        Inside Div Two: {{ aThirdProperty }}
        <div ng-init="aThirdProperty = 'data for 3rd property'"
             ng-controller="SomeController">
            Inside Div Three: {{ aThirdProperty }}
            <div ng-init="aFourthProperty">
                Inside Div Four: {{ aThirdProperty }}
            </div>
        </div>
    </div>
</div>
```

如果直接运行这段代码会报错，因为没有在JavaScript中定义所需的控制器，下面就来定义这个控制器：

```
angular.module('myApp', [])
.controller('SomeController', function($scope) {
    // 可以留空，但需要被定义
})
```

刷新页面，会发现第二个div中由于{{ aTgirdProperty }}未定义，因此什么都没有输出。第三个div显示了设置在继承来的作用域中的data for a 3rd property，如图10-1所示。

为了进一步证明作用域的继承机制是向下而非向上进行的，下面再看另外一个例子，展示的是{{ aThirdProperty }}从父作用域继承而来：

```
<div ng-app="myApp"
    ng-init="someProperty = 'some data'"></div>
<div ng-init="siblingProperty='moredata'">
    Inside Div Two: {{ aThirdProperty }}
    <div ng-init="aThirdProperty = 'data for 3rd property'"
        ng-controller="SomeController">
```

```
        Inside Div Three: {{ aThirdProperty }}
        <div ng-controller="SecondController">
            Inside Div Four: {{ aThirdProperty }}
        </div>
    </div>
</div>
```

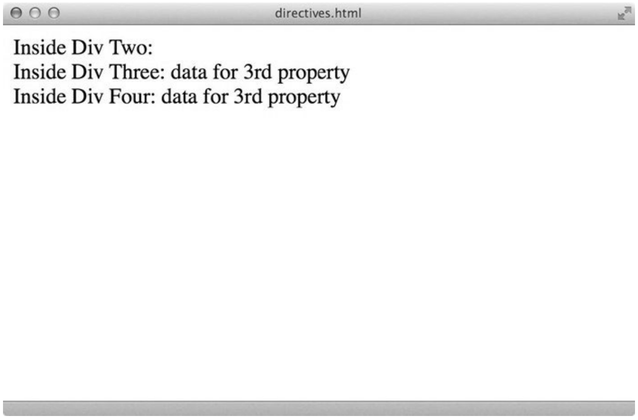


图10-1 指令

在JavaScript中加入SecondController的定义：

```
angular.module('myApp', [])
.controller('SomeController', function($scope) {
    // 可以留空，但需要被定义
})
.controller('SecondController', function($scope) {
    // 同样可以留空
})
```

如果要创建一个能够从外部原型继承作用域的指令，将scope属性设置为true：

```
angular.module('myApp', [])
.directive('myDirective', function() {
    return {
        restrict: 'A',
        scope: true
    };
});
```

下面用指令来改变DOM的作用域：

```
<div ng-app="myApp"
    ng-init="someProperty = 'some data'"></div>
<div ng-init="siblingProperty='moredata'">
    Inside Div Two: {{ aThirdProperty }}
    <div ng-init="aThirdProperty = 'data for 3rd property'"
        ng-controller="SomeController">
        Inside Div Three: {{ aThirdProperty }}
        <div ng-controller="SecondController">
            Inside Div Four: {{ aThirdProperty }}
            <br>
            Outside myDirective: {{ myProperty }}
            <div my-directive ng-init="myProperty = 'wow, this is cool'">
                Inside myDirective: {{ myProperty }}
```

10

```
            <div>
        </div>
    </div>
</div>
```

在线示例：http://jsbin.com/ITEBAF/1/edit。

现在，我们已经理解了指令的外部作用域和继承作用域是如何工作的，下面介绍最后一个和作用域相关的难题：隔离作用域。

10.2.2　隔离作用域

隔离作用域可能是scope属性三个选项中最难理解的一个，但也是最强大的。隔离作用域的概念是以面向对象编程为基础的。AngularJS指令的作用域中可以看到如Small Talk语言和SOLID原则的影子。

具有隔离作用域的指令最主要的使用场景是创建可复用的组件，组件可以在未知上下文中使用，并且可以避免污染所处的外部作用域或不经意地污染内部作用域。

创建具有隔离作用域的指令需要将scope属性设置为一个空对象{}。如果这样做了，指令的模板就无法访问外部作用域了：

```
<div ng-controller='MainController'>
    Outside myDirective: {{ myProperty }}
    <div my-directive ng-init="myProperty = 'wow, this is cool'">
        Inside myDirective: {{ myProperty }}
    </div>
</div>

angular.module('myApp', [])
.controller('MainController', function($scope) {
})
.directive('myDirective', function() {
    return {
        restrict: 'A',
        scope: {},
        priority: 100,
        template: '<div>Inside myDirective {{ myProperty }}</div>'
    };
});
```

注意，这里为myDirective设置了一个高优先级。由于ngInit指令会以非零的优先级运行，这个例子将会优先运行ngInit指令，然后才是我们定义的指定，并且这个myProperty在$scope对象中是有效的。

在线示例：http://jsbin.com/eguDEpa/1/edit。

示例代码的效果与将scope设置为true几乎是相同的。下面看一下使用继承作用域的指令的例子，对比一下二者：

```
<div ng-init="myProperty='wow,thisiscool'">
    Surrounding scope: {{ myProperty }}
    <div my-inherit-scope-directive></div>
    <div my-directive></div>
</div>
```

JavaScript代码：

```
angular.module('myApp', [])
.directive('myDirective', function() {
    return {
        restrict: 'A',
        template: 'Inside myDirective, isolate scope: {{ myProperty }}',
        scope: {}
    };
})
.directive('myInheritScopeDirective', function() {
    return {
        restrict: 'A',
        template: 'Inside myDirective, isolate scope: {{ myProperty }}',
        scope: true
    };
});
```

在线示例：http://jsbin.com/OxAlek/1/edit。

理解了最重要的关于作用域的概念后，就可以将隔离作用域中的属性同外部世界进行绑定，使得隔离作用域可以和外部进行交互。

10.3 绑定策略

使用无数据的隔离作用域并不常见。AngularJS提供了几种方法能够将指令内部的隔离作用域，同指令外部的作用域进行数据绑定。

为了让新的指令作用域可以访问当前本地作用域中的变量，需要使用下面三种别名中的一种。

本地作用域属性：使用@符号将本地作用域同DOM属性的值进行绑定。指令内部作用域可以使用外部作用域的变量：


```
@ (or @attr)
```

现在，可以在指令中使用绑定的字符串了。

双向绑定：通过=可以将本地作用域上的属性同父级作用域上的属性进行双向的数据绑定。就像普通的数据绑定一样，本地属性会反映出父数据模型中所发生的改变。

```
= (or =attr)
```

父级作用域绑定 通过&符号可以对父级作用域进行绑定，以便在其中运行函数。意味着对这个值进行设置时会生成一个指向父级作用域的包装函数。

要使调用带有一个参数的父方法，我们需要传递一个对象，这个对象的键是参数的名称，值是要传递给参数的内容。

```
& (or &attr)
```

例如，假设我们在开发一个电子邮件客户端，并且要创建一个电子邮件的文本输入框：

```
<input type="text" ng-model="to"/>
<!-- 调用指令 -->
<div scope-example ng-model="to"
    on-send="sendMail(email)"
    from-name="ari@fullstack.io" />
```

这里有一个数据模型（ng-model），一个函数（sendMail()）和一个字符串（from-name）。在指令中做如下设置以访问这些内容：

```
scope: {
    ngModel: '=', // 将ngModel同指定对象绑定
    onSend: '&',  // 将引用传递给这个方法
    fromName: '@' // 储存与fromName相关联的字符串
}
```

双向数据绑定

双向数据绑定或许是AngularJS中最强大的功能，借助它我们可以将一个私有作用域中的属性同DOM中的属性值进行绑定。在前面一章已经有一个很好的例子，展示了ng-model如何在外部世界同自定义的指令之间进行双向数据绑定，这个指令在很多方面都同ng-bind相似。回顾一下上一章的内容，并做相应的练习，来更好地理解这个重要的概念。

10.3.1　transclude

transclude是一个可选的参数。如果设置了，其值必须为true，它的默认值是false。

嵌入有时被认为是一个高级主题，但某些情况下它与我们刚刚学习过的作用域之间会有非常好的配合。使用嵌入也会很好地扩充我们的工具集，特别是在创建可以在团队、项目、AngularJS社区之间共享的HTML代码片段时。

嵌入通常用来创建可复用的组件，典型的例子是模态对话框或导航栏。

我们可以将整个模板，包括其中的指令通过嵌入全部传入一个指令中。这样做可以将任意内容和作用域传递给指令。transclude参数就是用来实现这个目的的，指令的内部可以访问外部指令的作用域，并且模板也可以访问外部的作用域对象。

为了将作用域传递进去，scope参数的值必须通过{}或true设置成隔离作用域。如果没有设置scope参数，那么指令内部的作用域将被设置为传入模板的作用域。

只有当你希望创建一个可以包含任意内容的指令时，才使用transclude: true。

嵌入允许指令的使用者方便地提供自己的HTML模板，其中可以包含独特的状态和行为，并对指令的各方面进行自定义。

下面一起来实现个小例子，创建一个可以被自定义的可复用指令。

我们来创建一个可以复用的侧边栏，同WordPress博客的侧边栏很相似。我们希望可以保持CSS样式的一致性，同时又希望可以在复用时尽量少写HTML代码。

例如，假设我们想创建一个包括标题和少量HTML内容的侧边栏，如下所示：

```
<div sideboxtitle="Links">
    <ul>
        <li>First link</li>
        <li>Second link</li>
    </ul>
</div>
```

为这个侧边栏创建一个简单的指令，并将transclude参数设置为true：

```
angular.module('myApp', [])
.directive('sidebox', function() {
    return {
        restrict: 'EA',
        scope: {
            title: '@'
        },
        transclude: true,
        template: '<div class="sidebox">\
            <div class="content">\
                <h2 class="header">{{ title }}</h2>\
                <span class="content" ng-transclude>\
                </span>\
            </div>\
        </div>'
    };
});
```

这段代码告诉AngularJS编译器，将它从DOM元素中获取的内容放到它发现ng-transclude指令的地方。

借助transclusion，我们可以将指令复用到第二个元素上，而无须担心样式和布局的一致性问题。

例如，下面的代码会产生两个样式完全一致的侧边栏，效果如图10-2所示。

```
<div sideboxtitle="Links">
    <ul>
        <li>First link</li>
        <li>Second link</li>
    </ul>
</div>
<div sideboxtitle="TagCloud">
    <div class="tagcloud">
        <a href="">Graphics</a>
        <a href="">AngularJS</a>
        <a href="">D3</a>
        <a href="">Front-end</a>
        <a href="">Startup</a>
    </div>
</div>
```

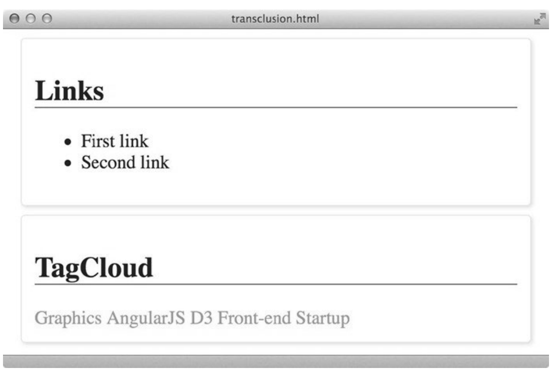


图10-2 侧边栏

10

如果指令使用了transclude参数，那么在控制器（下面马上会介绍）中就无法正常监听数据模型的变化了。这就是最佳实践总是建议在链接函数里使用$watch服务的原因。

10.3.2 controller（字符串或函数）

controller参数可以是一个字符串或一个函数。当设置为字符串时，会以字符串的值为名字，来查找注册在应用中的控制器的构造函数：

```
angular.module('myApp', [])
.directive('myDirective',  function() {
    restrict: 'A', // 始终需要
    controller: 'SomeController'
})

// 应用中其他的地方，可以是同一个文件或被index.html包含的另一个文件
angular.module('myApp')
.controller('SomeController', function($scope, $element, $attrs, $transclude) {
    // 控制器逻辑放在这里
});
```

可以在指令内部通过匿名构造函数的方式来定义一个内联的控制器：

```
angular.module('myApp',[])
.directive('myDirective', function() {
    restrict: 'A',
    controller:
    function($scope, $element, $attrs, $transclude) {
        // 控制器逻辑放在这里
    }
});
```

我们可以将任意可以被注入的AngularJS服务传递给控制器。例如，如果我们想要将$log服务传入控制器，只需简单地将它注入到控制器中，便可以在指令中使用它了。

控制器中也有一些特殊的服务可以被注入到指令当中。这些服务有：

1. $scope

与指令元素相关联的当前作用域。

2. $element

当前指令对应的元素。

3. $attrs

由当前元素的属性组成的对象。例如，下面的元素：

```
<div id="aDiv"class="box"></div>
```

具有如下的属性对象：

```
{
 id: "aDiv",
 class: "box"
}
```

4. $transclude

嵌入链接函数会与对应的嵌入作用域进行预绑定。

transclude链接函数是实际被执行用来克隆元素和操作DOM的函数。

> 在控制器内部操作DOM是和AngularJS风格相悖的做法，但通过链接函数就可以实现这个需求。仅在compile参数中使用transcludeFn是推荐的做法。

例如，我们想要通过指令来添加一个超链接标签。可以在控制器内的$transclude函数中实现，如下所示：

```
angular.module('myApp')
.directive('link', function() {
    return {
        restrict: 'EA',
        transclude: true,
        controller:
            function($scope, $element, $transclude, $log) {
            $transclude(function(clone) {
                var a = angular.element('<a>');
                a.attr('href', clone.text());
                a.text(clone.text());
                $log.info("Created new a tag in link directive");
                $element.append(a);
            });
        }
    };
});
```


指令的控制器和link函数可以进行互换。控制器主要是用来提供可在指令间复用的行为，但链接函数只能在当前内部指令中定义行为，且无法在指令间复用。

> link函数可以将指令互相隔离开来，而controller则定义可复用的行为。

由于指令可以require其他指令所使用的控制器，因此控制器常被用来放置在多个指令间共享的动作。

如果我们希望将当前指令的API暴露给其他指令使用，可以使用controller参数，否则可以使用link来构造当前指令元素的功能性。如果我们使用了scope.$watch()或者想要与DOM元素做实时的交互，使用链接会是更好的选择。

技术上讲，$scope会在DOM元素被实际渲染之前传入到控制器中。在某些情况下，例如使用了嵌入，控制器中的作用域所反映的作用域可能与我们所期望的不一样，这种情况下，$scope对象无法保证可以被正常更新。

> 当想要同当前屏幕上的作用域交互时，可以使用被传入到link函数中的scope参数。

10.3.3 controllerAs（字符串）

controllerAs参数用来设置控制器的别名，可以以此为名来发布控制器，并且作用域可以访问controllerAs。这样就可以在视图中引用控制器，甚至无需注入$scope。

例如，创建一个MainController，然后不要注入$scope，如下所示：

```
angular.module('myApp')
.controller('MainController', function() {
    this.name = "Ari";
});
```

现在，在HTML中无需引用作用域就可以使用MainController。

```
<div ng-appng-controller="MainControllerasmain">
    <input type="text" ng-model="main.name" />
    <span>{{ main.name }}</span>
</div>
```

这个参数看起来好像没什么大用，但它给了我们可以在路由和指令中创建匿名控制器的强大能力。这种能力可以将动态的对象创建成为控制器，并且这个对象是隔离的、易于测试的。

例如，可以在指令中创建匿名控制器，如下所示：

```
angular.module('myApp')
.directive('myDirective', function() {
    return {
        restrict: 'A',
        template: '<h4>{{ myController.msg }}</h4>',
        controllerAs: 'myController',
        controller: function() {
            this.msg = "Hello World"
        }
    };
});
```

10.3.4 require（字符串或数组）

require参数可以被设置为字符串或数组，字符串代表另外一个指令的名字。require会将控制器注入到其值所指定的指令中，并作为当前指令的链接函数的第四个参数。

字符串或数组元素的值是会在当前指令的作用域中使用的指令名称。

scope会影响指令作用域的指向，是一个隔离作用域，一个有依赖的作用域或者完全没有作用域。在任何情况下，AngularJS编译器在查找子控制器时都会参考当前指令的模板。

如果不使用^前缀，指令只会在自身的元素上查找控制器。

```
//...
restrict: 'EA',
require: 'ngModel'
//...
```

指令定义只会查找定义在指令作当前用域中的ng-model=""。

```
<!-- 指令会在本地作用域查找ng-model -->
<div my-directive ng-model="object"></div>
```

require参数的值可以用下面的前缀进行修饰，这会改变查找控制器时的行为：

?

如果在当前指令中没有找到所需要的控制器，会将null作为传给link函数的第四个参数。

^

如果添加了^前缀，指令会在上游的指令链中查找require参数所指定的控制器。

?^

将前面两个选项的行为组合起来，我们可选择地加载需要的指令并在父指令链中进行查找。

没有前缀

如果没有前缀，指令将会在自身所提供的控制器中进行查找，如果没有找到任何控制器（或具有指定名字的指令）就抛出一个错误。

10.4 AngularJS 的生命周期

在AngularJS应用起动前，它们以HTML文本的形式保存在文本编辑器中。应用启动后会进行编译和链接，作用域会同HTML进行绑定，应用可以对用户在HTML中进行的操作进行实时响应。这个神器的效果是如何发生的？创建高效率的应用需要了解什么？

在这个过程中总共有两个主要阶段。

10.4.1 编译阶段

第一个阶段是编译阶段。在编译阶段，AngularJS会遍历整个HTML文档并根据JavaScript中的指令定义来处理页面上声明的指令。

每一个指令的模板中都可能含有另外一个指令，另外一个指令也可能会有自己的模板。当AngularJS调用HTML文档根部的指令时，会遍历其中所有的模板，模板中也可能包含带有模板的指令。

> 模板树可能又大又深，但有一点需要注意，尽管元素可以被多个指令所支持或修饰，这些指令本身的模板中也可以包含其他指令，但只有属于最高优先级指令的模板会被解析并添加到模板树中。这里有一个建议，就是将包含模板的指令和添加行为的指令分离开来。如果一个元素已经有一个含有模板的指令了，永远不要对其用另一个指令进行修饰。只有具有最高优先级的指令中的模板会被编译。

一旦对指令和其中的子模板进行遍历或编译，编译后的模板会返回一个叫做模板函数的函数。我们有机会在指令的模板函数被返回前，对编译后的DOM树进行修改。

在这个时间点DOM树还没有进行数据绑定，意味着如果此时对DOM树进行操作只会有很少的性能开销。基于此点，ng-repeat和ng-transclude等内置指令会在这个时候，也就是还未与任何作用域数据进行绑定时对DOM进行操作。

以ng-repeat为例，它会遍历指定的数组或对象，在数据绑定之前构建出对应的DOM结构。

如果我们用ng-repeat来创建无序列表，其中的每一个<li>都会被ng-click指令所修饰，这个过程会使得性能比手动创建列表要快得多，尤其是列表中含有上百个元素时。

与克隆<li>元素，再将其与数据进行链接，然后对每个元素都循环进行此操作的过程不同，我们仅需要先将无需列表构造出来，然后将新的DOM（编译后的DOM）传递给指令生命周期中的下一个阶段，即链接阶段。

一个指令的表现一旦编译完成，马上就可以通过编译函数对其进行访问，编译函数的签名包含有访问指令声明所在的元素（tElemente）及该元素其他属性（tAttrs）的方法。这个编译函数返回前面提到的模板函数，其中含有完整的解析树。

这里的重点是，由于每个指令都可以有自己的模板和编译函数，每个模板返回的也都是自己的模板函数。链条顶部的指令会将内部子指令的模板合并在一起成为一个模板函数并返回，但在树的内部，只能通过模板函数访问其所处的分支。

最后，模板函数被传递给编译后的DOM树中每个指令定义规则中指定的链接函数，

10.4.2　compile（对象或函数）

compile选项可以返回一个对象或函数。

理解compile和link选项是AngularJS中需要深入讨论的高级话题之一，对于了解AngularJS究竟是如何工作的至关重要。

compile选项本身并不会被频繁使用，但是link函数则会被经常使用。本质上，当我们设置了link选项，实际上是创建了一个postLink()链接函数，以便compile()函数可以定义链接函数。

通常情况下，如果设置了compile函数，说明我们希望在指令和实时数据被放到DOM中之前进行DOM操作，在这个函数中进行诸如添加和删除节点等DOM操作是安全的。

> compile和link选项是互斥的。如果同时设置了这两个选项，那么会把compile所返回的函数当作链接函数，而link选项本身则会被忽略。

```
// ...
compile: function(tEle, tAttrs, transcludeFn) {
    var tplEl = angular.element('<div>' +
        '<h2></h2>' +
        '</div>');
        var h2 = tplEl.find('h2');
        h2.attr('type', tAttrs.type);
        h2.attr('ng-model', tAttrs.ngModel);
        h2.val("hello");
        tEle.replaceWith(tplEl);
        return function(scope, ele, attrs) {
        // 连接函数
    };
}
//...
```

如果模板被克隆过，那么模板实例和链接实例可能是不同的对象。因此在编译函数内部，我

们只能转换那些可以被安全操作的克隆DOM节点。不要进行DOM事件监听器的注册：这个操作应该在链接函数中完成。

编译函数负责对模板DOM进行转换。

链接函数负责将作用域和DOM进行链接。在作用域同DOM链接之前可以手动操作DOM。在实践中，编写自定义指令时这种操作是非常罕见的，但有几个内置指令提供了这样的功能。了解这个流程对于理解AngularJS真正的工作方式很有帮助。

10.4.3　链接

用`link`函数创建可以操作DOM的指令。

链接函数是可选的。如果定义了编译函数，它会返回链接函数，因此当两个函数都定义了时，编译函数会重载链接函数。如果我们的指令很简单，并且不需要额外的设置，可以从工厂函数（回调函数）返回一个函数来代替对象。如果这样做了，这个函数就是链接函数。

下面两种定义指令的方式在功能上是完全一样的：

```
angular.module('myApp', [])
.directive('myDirective', function() {
    return {
        pre: function(tElement, tAttrs, transclude) {
            // 在子元素被链接之前执行
            // 在这里进行Don转换不安全
            // 之后调用'lihk'h函数将无法定位要链接的元素
    },
    post: function(scope, iElement, iAttrs, controller) {
            // 在子元素被链接之后执行
            // 如果在这里省略掉编译选项
            //在这里执行DOM转换和链接函数一样安全吗
        }
    };
});

angular.module('myApp', [])
.directive('myDirective', function() {
    return {
        link: function(scope, ele, attrs) {
            return {
                pre: function(tElement, tAttrs, transclude) {
            // 在子元素被链接之前执行
            // 在这里进行Don转换不安全
            // 之后调用'lihk'h函数将无法定位要链接的元素
            },
            post: function(scope, iElement, iAttrs, controller) {
            // 在子元素被链接之后执行
            // 如果在这里省略掉编译选项
            //在这里执行DOM转换和链接函数一样安全吗
            }
        }
    }
});
```

当定义了编译函数来取代链接函数时，链接函数是我们能提供给返回对象的第二个方法，也就是`postLink`函数。本质上讲，这个事实正说明了链接函数的作用。它会在模板编译并同作用域

进行链接后被调用，因此它负责设置事件监听器，监视数据变化和实时的操作DOM。

link函数对绑定了实时数据的DOM具有控制能力，因此需要考虑性能问题。回顾一下10.4节中关于性能的考虑，在选择是用编译函数还是链接函数实现功能时，将性能影响考虑进去。

链接函数的签名如下：

```
link: function(scope, element, attrs) {
    // 在这里操作DOM
}
```

如果指令定义中有require选项，函数签名中会有第四个参数，代表控制器或者所依赖的指令的控制器。

```
// require 'SomeController',
link: function(scope, element, attrs, SomeController) {
    // 在这里操作DOM，可以访问required指定的控制器
}
```

如果require选项提供了一个指令数组，第四个参数会是一个由每个指令所对应的控制器组成的数组。

下面看一下链接函数中的参数：

scope

指令用来在其内部注册监听器的作用域。

iElement

iElement参数代表实例元素，指使用此指令的元素。在postLink函数中我们应该只操作此元素的子元素，因为子元素已经被链接过了。

iAttrs

iAttrs参数代表实例属性，是一个由定义在元素上的属性组成的标准化列表，可以在所有指令的链接函数间共享。会以JavaScript对象的形式进行传递。

controller

controller参数指向require选项定义的控制器。如果没有设置require选项，那么controller参数的值为undefined。

控制器在所有的指令间共享，因此指令可以将控制器当作通信通道（公共API）。如果设置了多个require，那么这个参数会是一个由控制器实例组成的数组，而不只是一个单独的控制器。

10.5　ngModel

ngModel是一个用法特殊的指令，它提供更底层的API来处理控制器内的数据。当我们在指令中使用ngModel时能够访问一个特殊的API，这个API用来处理数据绑定、验证、CSS更新等不实际操作DOM的事情。

ngModel控制器会随ngModel被一直注入到指令中，其中包含了一些方法。为了访问ngModelController必须使用require设置（像前面的例子中那样）：

```
angular.module('myApp')
.directive('myDirective',function(){
    return {
        require: '?ngModel',
        link: function(scope, ele, attrs, ngModel) {
            if (!ngModel) return;
            // 现在我们的指令中已经有ngModelController的一个实例
        }
    };
});
```

如果不设置require选项，ngModelController就不会被注入到指令中。

注意，这个指令没有隔离作用域。如果给这个指令设置隔离作用域，将导致内部ngModel无法更新外部ngModel的对应值：AngularJS会在本地作用域以外查询值。

为了设置作用域中的视图值，需要调用ngModel.$setViewValue()函数。ngModel.$setViewValue()函数可以接受一个参数。

value（字符串）：value参数是我们想要赋值给ngModel实例的实际值。这个方法会更新控制器上本地的$viewValue，然后将值传递给每一个$parser函数（包括验证器）。

当值被解析，且$parser流水线中所有的函数都调用完成后，值会被赋给$modelValue属性，并且传递给指令中ng-model属性提供的表达式。

最后，所有步骤都完成后，$viewChangeListeners中所有的监听器都会被调用。

注意，单独调用$setViewValue()不会唤起一个新的digest循环，因此如果想更新指令，需要在设置$viewValue后手动触发digest。

$setViewValue()方法适合于在自定义指令中监听自定义事件（比如使用具有回调函数的jQuery插件），我们会希望在回调时设置$viewValue并执行digest循环。

```
angular.module('myApp')
.directive('myDirective', function() {
    return {
        require: '?ngModel',
        link: function(scope, ele, attrs, ngModel) {
            if (!ngModel) return;

            $(function() {
                ele.datepicker({
                    onSelect: function(date) {
                        // 设置视图和调用apply
                        scope.$apply(function() {
                            ngModel.$setViewValue(date);
                        });
                    }
                });
            });
        }
    };
});
```

10.5.1 自定义渲染

在控制器中定义$render方法可以定义视图具体的渲染方式。这个方法会在$parser流水线完成后被调用。

由于这个方法会破坏AngularJS的标准工作方式，因此一定要谨慎使用：

```
angular.module('myApp')
.directive('myDirective', function() {
    return {
        require: '?ngModel',
        link: function(scope, ele, attrs, ngModel) {
            if (!ngModel) return;

            ngModel.$render = function() {
                element.html(ngModel.$viewValue() || 'None');
            };
        }
    };
});
```

10.5.2 属性

ngModelController中有几个属性可以用来检查甚至修改视图。

1. $viewValue

$viewValue属性保存着更新视图所需的实际字符串。

2. $modelValue

$modelValue由数据模型持有。$modelValue和$viewValue可能是不同的，取决于$parser流水线是否对其进行了操作。

3. $parsers

$parsers的值是一个由函数组成的数组，其中的函数会以流水线的形式被逐一调用。ngModel从DOM中读取的值会被传入$parsers中的函数，并依次被其中的解析器处理。

这是为了对值进行处理和修饰。我们已经简单介绍过验证流水线是如何工作的了。更多关于通过创建$parser来进行验证的信息请查阅10.6节。

4. $formatters

$formatters的值是一个由函数组成的数组，其中的函数会以流水线的形式在数据模型的值发生变化时被逐一调用。它和$parser流水线互不影响，用来对值进行格式化和转换，以便在绑定了这个值的控件中显示。

5. $viewChangeListeners

$viewChangeListeners的值是一个由函数组成的数组，其中的函数会以流水线的形式在视图中的值发生变化时被逐一调用。通过$viewChangeListeners，可以在无需使用$watch的情况下实现类似的行为。由于返回值会被忽略，因此这些函数不需要返回值。

6. $error

$error对象中保存着没有通过验证的验证器名称以及对应的错误信息。

7. $pristine

$pristine的值是布尔型的，可以告诉我们用户是否对控件进行了修改。

8. $dirty

$dirty的值和$pristine相反，可以告诉我们用户是否和控件进行过交互。

9. $valid

$valid值可以告诉我们当前的控件中是否有错误。当有错误时值为false，没有错误时值为true。

10. $invalid

$invalid值可以告诉我们当前控件中是否存在至少一个错误，它的值和$valid相反。

10.6 自定义验证

前面验证章节介绍过如何使用指令创建自定义验证。使用AngularJS可以方便地通过指令添加自定义验证。例如，我们需要验证username在数据库中是否合法，可以实现一个指令，用来在表单发生变化时发送Ajax请求：

```
angular.module('validationExample', [])
.directive('ensureUnique',function($http) {
    return {
        require: 'ngModel',
        link: function(scope, ele, attrs, c) {
            scope.$watch(attrs.ngModel, function() {
                $http({
                    method: 'POST',
                    url: '/api/check/' + attrs.ensureUnique,
                    data: {field: attrs.ensureUnique, valud:scope.ngModel
                }).success(function(data,status,headers,cfg) {
                    c.$setValidity('unique', data.isUnique);
                }).error(function(data,status,headers,cfg) {
                    c.$setValidity('unique', false);
                });
            });
        }
    };
});
```

出于演示目的，尽管我们在指令内置入了一个$http调用，但是在产品中的指令内使用$http是不明智的。相反，将它置入到服务中会更好。关于服务的更多信息请参第14章。

可以像使用其他AngularJS的内置验证一样来使用这个自定义验证，如下所示：

```
<input type="text"
       placeholder="Desired username"
       name="username"
```

```
ng-model="signup.username"
ng-minlength="3"
ng-maxlength="20"
ensure-unique="username" required />
```

在这个自定义验证中，每当ngModel中对应的字段发生变化就会向服务器发送请求，以检查用户名是否是唯一的。

第 11 章

AngularJS模块加载

AngularJS模块可以在被加载和执行之前对其自身进行配置。我们可以在应用的加载阶段应用不同的逻辑组。

11.1 配置

在模块的加载阶段，AngularJS会在提供者注册和配置的过程中对模块进行配置。在整个AngularJS的工作流中，这个阶段是唯一能够在应用启动前进行修改的部分。

```
angular.module('myApp', [])
    .config(function($provide) {
    });
```

这本书的大部分内容都在使用config()函数的语法糖，并在配置阶段执行。例如，我们在某个模块之上创建一个服务或指令时：

```
angular.module('myApp', [])
.factory('myFactory', function(){
    var service = {};
    return service;
})
.directive('myDirective', function(){
    return {
        template: '<button>Click me</button>'
    }
})
```

AngularJS会在编译时执行这些辅助函数。它们在功能上等同于下面的写法：

```
angular.module('myApp', [])
.config(function($provide ,$compileProvider) {
    $provide.factory('myFactory', function() {
        var service = {};
        return service;
    });
    $compileProvider.directive('myDirective', function() {
        return {
            template: '<button>Click me</button>'
        };
    });
});
```

需要特别注意，AngularJS会以这些函数书写和注册的顺序来执行它们。也就是说我们无法

注入一个尚未注册的提供者。

> **唯一**例外的是constant()方法，这个方法总会在所有配置块之前被执行。

当对模块进行配置时，需要格外注意只有少数几种类型的对象可以被注入到config()函数中：提供者和常量。如果我们将一个服务注入进去，会在真正对其进行配置之前就意外地把服务实例化了。

这种对配置服务进行严格限制的另外一个副作用就是，我们只能注入用provider()语法构建的服务，其他的则不行。

更多关于用provider()语法构建服务的内容，请查看第14章。

这些config()代码块可以对我们的服务进行自定义配置，例如设置API密钥或自定义URL等。

也可以定义多个配置块，它们会按照顺序执行，这样就可以将应用不同阶段的配置代码集中在不同的代码块中。

```
angular.module('myApp', [])
.config(function($routeProvider) {
    $routeProvider.when('/', {
        controller: 'WelcomeController',
        template: 'views/welcome.html'
    });
})
.config(function(ConnectionProvider) {
    ConnectionProvider.setApiKey('SOME_API_KEY');
});
```

config()函数接受一个参数。

- configFunction（函数）：AngularJS在模块加载时会执行这个函数。

11.2　运行块

和配置块不同，运行块在注入器创建之后被执行，它是所有AngularJS应用中第一个被执行的方法。

运行块是AngularJS中与main方法最接近的概念。运行块中的代码块通常很难进行单元测试，它是和应用本身高度耦合的。

运行块通常用来注册全局的事件监听器。例如，我们会在.run()块中设置路由事件的监听器以及过滤未经授权的请求。

假设我们需要在每次路由发生变化时，都执行一个函数来验证用户的权限，放置这个功能唯一合理的地方就是run方法：

```
angular.module('myApp', [])
.run(function($rootScope, AuthService) {
    $rootScope.$on('$routeChangeStart', function(evt, next, current) {
        // 如果用户未登录
        if (!AuthService.userLoggedIn()) {
```

```
            if (next.templateUrl === "login.html") {
                // 已经转向登录路由因此无需重定向
            } else {
                $location.path('/login');
            }
        }
    });
});
```

run()函数接受个参数。

- initializeFn（函数）AngularJS在注入器创建后会执行这个函数。

第 12 章

多重视图和路由

能够从页面的一个视图跳转到另外一个视图，对单页面应用来讲是至关重要的。当应用变得越来越复杂时，我们需要一个合理的方式来管理用户在使用过程中看到的界面。

也许我们已经通过在主HTML中添加行内的模板代码，实现了对视图的管理，但这些代码也会变得越来越复杂和难以管理，同时其他开发者也很难加入到开发工作中来。另外，由于应用的状态信息会包含在URL中，我们也无法将代码直接复制并发送给朋友。

除了用ng-include指令在视图中引用多个模板外，更好的做法是将视图分解成布局和模板视图，并且根据用户当前访问的URL来展示对应的视图。

我们会将这些模板分解到视图中，并在布局模板内进行组装。AngularJS允许我们在$route服务的提供者$routeProvider中通过声明路由来实现这个功能。

通过$routeProvider，可以发挥出浏览器历史导航的优势，并让用户基于浏览器当前的URL地址创建书签或分享页面。

12.1 安装

从1.2版本开始，AngularJS将ngRoutes从核心代码中剥离出来成为独立的模块。我们需要安装并引用它，才能够在AngularJS应用中正常地使用路由功能。

可以从code.angularjs.org下载它，然后保存到一个可以在HTML页面中进行引用的位置，例如js/vendor/angular-route.js。

也可以用Bower来安装，这样会将它存放到Bower的目录中。查看34.6节获取更多关于Bower的信息。

```
$bower install--save angular -route
```

在HTML中，需要在AngularJS之后引用angular-route：

```
<script src="js/vendor/angular.js"></script>
<script src="js/vendor/angular-route.js"></script>
```

最后，要把ngRoute模块在我们的应用中当作依赖加载进来：

```
angular.module('myApp', ['ngRoute']);
```

12.2 布局模板

要创建一个布局模板，需要修改HTML以告诉AngularJS把模板渲染到何处。通过将ng-view指令和路由组合到一起，我们可以精确地指定当前路由所对应的模板在DOM中的渲染位置。

例如，布局模板看起来可能是下面这样的：

```
<header>
    <h1>Header</h1>
</header>
<div class="content">
    <div ng-view></div>
</div>
<footer>
    <h5>Footer</h5>
</footer>
```

这个例子中，我们将所有需要渲染的内容都放到了<div class="content">中，而<header>和<footer>中的内容在路由改变时不会有任何变化。

ng-view是由ngRoute模块提供的一个特殊指令，它的独特作用是在HTML中给$route对应的视图内容占位。

它会创建自己的作用域并将模板嵌套在内部。

> mg-view是一个优先级为1000的终极指令。AngularJS不会运行同一个元素上的低优先级指令（例如<div ng-view></div>元素上其他指令都是没有意义的）。

ngView指令遵循以下规则。

- 每次触发$routeChangeSuccess事件，视图都会更新。
- 如果某个模板同当前的路由相关联：
  - 创建一个新的作用域；
  - 移除上一个视图，同时上一个作用域也会被清除；
  - 将新的作用域同当前模板关联在一起；
  - 如果路由中有相关的定义，那么就把对应的控制器同当前作用域关联起来；
  - 触发$viewContentLoaded事件；
  - 如果提供了onload属性，调用该属性所指定的函数。

12

12.3 路由

我们可以使用AngularJS提供的when和otherwise两个方法来定义应用的路由。

用config函数在特定的模块或应用中定义路由。

```
angular.module('myApp', []).
    config(['$routeProvider', function($routeProvider) {
        // 在这里定义路由
    }]);
```

这里我们用了数组这种特殊的依赖注入语法。查看第13章获取更多关于这种语法的内容。

现在，我们可以用when方法来添加一个特定的路由。这个方法可以接受两个参数（when(path, route)）。

下面的例子展示了如何创建一个独立的路由：

```
angular.module('myApp', []).
    config(['$routeProvider', function($routeProvider) {
        $routeProvider
            .when('/', {
                templateUrl: 'views/home.html',
                controller: 'HomeController'
            });
    }]);
```

第一个参数是路由路径，这个路径会与$location.path进行匹配，$location.path也就是当前URL的路径。如果路径后面还有其他内容，或使用了双斜线也可以正常匹配。我们可以在URL中存储参数，参数需要以冒号开头（例如:name），后面会讨论如何用$routeParams读取这些参数。

第二个参数是配置对象，决定了当第一个参数中的路由能够匹配时具体做些什么。配置对象中可以进行设置的属性包括controller、template、templateURL、resolve、redirectTo和reloadOnSearch。

一个复杂的路由方案会包含多个路由，以及一个可以将所有意外路径进行重定向的捕获器。

```
angular.module('myApp', []).
    config(['$routeProvider', function($routeProvider) {
        $routeProvider
            .when('/', {
                templateUrl: 'views/home.html',
                controller: 'HomeController'
            })
            .when('/login', {
                templateUrl: 'views/login.html',
                controller: 'LoginController'
            })
            .when('/dashboard', {
                templateUrl: 'views/dashboard.html',
                controller: 'DashboardController',
                resolve: {
                    user: function(SessionService) {
                        return SessionService.getCurrentUser();
                    }
                }
            })
            .otherwise({
                redirectTo: '/'
            });
    }]);
```

1. controller

```
controller: 'MyController'
// 或者
```

```
controller: function($scope) {}
```

如果配置对象中设置了controller属性，那么这个指定的控制器会与路由所创建的新作用域关联在一起。如果参数值是字符型，会在模块中所有注册过的控制器中查找对应的内容，然后与路由关联在一起。如果参数值是函数型，这个函数会作为模板中DOM元素的控制器并与模板进行关联。

2. template

```
template: '<div><h2>Route</h2></div>'
```

AngularJS会将配置对象中的HTML模板渲染到对应的具有ng-view指令的DOM元素中。

3. templateUrl

```
templateUrl: 'views/template_name.html'
```

应用会根据templateUrl属性所指定的路径通过XHR读取视图（或者从$templateCache中读取）。如果能够找到并读取这个模板，AngularJS会将模板的内容渲染到具有ng-view指令的DOM元素中。

4. resolve

```
resolve: {
    'data': ['$http', function($http) {
        return $http.get('/api').then(
            function success(resp) { return response.data; },
            function error(reason) { return false; }
        );
    }];
}
```

如果设置了resolve属性，AngularJS会将列表中的元素都注入到控制器中。如果这些依赖是promise对象，它们在控制器加载以及$routeChangeSuccess被触发之前，会被resolve并设置成一个值。

列表对象可以是：

- 键，键值是会被注入到控制器中的依赖的名字；
- 工厂，即可以是一个服务的名字，也可以是一个返回值，它是会被注入到控制器中的函数或可以被resolve的promise对象。

在上面的例子中，resolve会发送一个$http请求，并将data的值替换为返回结果的值。列表中的键data会被注入到控制器中，所以在控制器中可以使用它。

5. redirectTo

```
redirectTo: '/home'
// 或者
redirectTo: function(route,path,search)
```

如果redirectTo属性的值是一个字符串，那么路径会被替换成这个值，并根据这个目标路径触发路由变化。

如果redirectTo属性的值是一个函数，那么路径会被替换成函数的返回值，并根据这个目

12

标路径触发路由变化。

如果redirectTo属性的值是一个函数，AngularJS会在调用它时传入下面三个参数中：

(1) 从当前路径中提取出的路由参数；

(2) 当前路径；

(3) 当前URL中的查询串。

6. reloadOnSearch

如果reloadOnSearch选项被设置为true（默认），当$location.search()发生变化时会重新加载路由。如果设置为false，那么当URL中的查询串部分发生变化时就不会重新加载路由。这个小窍门对路由嵌套和原地分页等需求非常有用。

现在介绍用when函数来设置路由。

下面的例子中设置了两个路由：一个首页路由和一个收件箱路由，同时首页路由被设置成默认路由。

```
angular.module('MyApp', []).
    config(['$routeProvider', function($routeProvider) {
        $routeProvider
            .when('/', {
                controller: 'HomeController',
                templateUrl: 'views/home.html'
            })
            .when('/inbox/:name', {
                controller: 'InboxController',
                templateUrl: 'views/inbox.html'
            })
            .otherwise({redirectTo: '/'});
      }]);
```

如上，我们已经用when方法设置了两个路由。otherwise方法会在没有任何路由匹配时被调用，我们用它设置了一个默认跳转到'/'路径的路由。

当浏览器加载AngularJS应用时，会将URL设置成默认路由所指向的路径。除非我们在浏览器中加载不同的URL，否则默认会使用/路由。

$routeParams

前面提到如果我们在路由参数的前面加上:，AngularJS就会把它解析出来并传递给$routeParams。例如，如果我们设置下面这样的路由：

```
$routeProvider
    .when('/inbox/:name', {
      controller: 'InboxController',
      templateUrl: 'views/inbox.html'
    });
```

AngularJS会在$routeParams中添加一个名为name的键，它的值会被设置为加载进来的URL中的值。

如果浏览器加载/inbox/all这个URL，那么$routeParams对象看起来会是下面这样：

```
{ name: 'all' }
```

需要注意，如果想要在控制器中访问这些变量，需要把$routeParams注入进控制器：

```
app.controller('InboxController', function($scope,$routeParams) {
    // 在这里访问$routeParams
});
```

12.4　$location 服务

AngularJS提供了一个服务用以解析地址栏中的URL，并让你可以访问应用当前路径所对应的路由。它同样提供了修改路径和处理各种形式导航的能力。

> $location服务对JavaScript中的window.location对象的API进行了更优雅地封装，并且和AngularJS集成在一起。

当应用需要在内部进行跳转时是使用$location服务的最佳场景，比如当用户注册后、修改或者登录后进行的跳转。

$location服务没有刷新整个页面的能力。如果需要刷新整个页面，需要使用$window.location对象（window.location的一个接口）。

1. path()

path()用来获取页面当前的路径：

```
$location.path(); // 返回当前路径
```

修改当前路径并跳转到应用中的另一个URL：

```
$location.path('/'); // 把路径修改为'/'路由
```

path()方法直接和HTML5的历史API进行交互，所以用户可通过点击后退按钮退回到上一个页面。

12

2. replace()

如果你希望跳转后用户不能点击后退按钮（对于登录之后的跳转这种发生在某个跳转之后的再次跳转很有用），AngularJS提供了replace()方法来实现这个功能：

```
$location.path('/home');
$location.replace();
// 或者
$location.path('/home').replace();
```

3. absUrl()

absUrl()方法用来获取编码后的完整URL：

```
$location.absUrl()
```

4. hash()

hash()方法用来获取URL中的hash片段：

```
$location.hash(); // 返回当前的hash片段
```

5. host()

host()方法用来获取URL中的主机：

```
$location.host();// 当前URL的主机
```

6. port()

port()方法用来获取URL中的端口号：

```
$location.port();// 当前URL的端口
```

7. protocol()

protocol()方法用来获取URL中的协议：

```
$location.protocol();// 当前URL的协议
```

8. search()

search()方法用来获取URL中的查询串：

```
$location.search();
```

我们可以向这个方法中传入新的查询参数，来修改URL中的查询串部分：

```
// 用对象设置查询
$location.search({name: 'Ari', username: 'auser'});
// 用字符串设置查询
$location.search('name=Ari&username=auser');
```

search方法可以接受两个参数。

- search（可选，字符串或对象）

这个参数代表新的查询参数。hash对象的值可以是数组。

- paramValue（可选，字符串）

如果search参数的类型是字符串，那么paramValue会做为该参数的值覆盖URL当中的对应值。如果paramValue的值是null，对应的参数会被移除掉。

9. url()

url()方法用来获取当前页面的URL：

```
$location.url(); // 该URL的字符串
```

如果调用url()方法时传了参数，会设置并修改当前的URL，这会同时修改URL中的路径、查询串和hash，并返回$location。

```
// 设置新的URL
$location.url('/home?name=Ari#hashthing');
```

url()方法可以接受两个参数。

- url（可选，字符串）

新的URL的基础的前缀。

❑ replace（可选，字符串）

想要修改成的路径。

12.5 路由模式

不同的路由模式在浏览器的地址栏中会以不同的URL格式呈现。$location服务默认会使用标签模式来进行路由。

路由模式决定你的站点的URL长成什么样子。

标签模式

标签（hashbang）是AngularJS用来同你的应用内部进行链接的技巧。标签模式是HTML5模式的降级方案，URL路径会以#符号开头。标签模式不需要重写标签，也不需要任何服务器端的支持。如果没有进行额外的指定，AngularJS将默认使用标签模式。

使用标签模式的URL看起来是这样的：

```
http://yoursite.com/#!/inbox/all
```

如果要显式指定配置并使用标签模式，需要在应用模块的config函数中进行配置：

```
angular.module('myApp', ['ngRoute'])
    .config(['$locationProvider', function($locationProvider) {
        $locationProvider.html5Mode(false);
    }]);
```

我们还可以配置hashPrefix，也就是标签模式下标签默认的前缀!符号。这个前缀也是AngularJS在比较老的浏览器中降级机制的一部分。这个符号是可以配置的：

```
angular.module('myApp', ['ngRoute'])
    .config(['$locationProvider', function($locationProvider) {
        $locationProvider.html5Mode(false);
        $locationProvider.hashPrefix('!');
    }]);
```

12

12.5.1 HTML5 模式

AngularJS支持的另外一种路由模式是html5模式。在这个模式中，URL看起来和普通的URL一样（在老式浏览器中看起来还是使用标签的URL）。例如，同样的路由在HTML5模式中看起来是这样的：

```
http://yoursite.com/inbox/all
```

在AngularJS内部，$location服务通过HTML5历史API让应用能够使用普通的URL路径来路由。当浏览器不支持HTML5历史API时，$location服务会自动使用标签模式的URL作为替代方案。

$location服务还有一个有趣的功能，当一个支持HTML5历史API的现代浏览器加载了一个带标签的URL时，它会为用户重写这个URL。

在HTML5模式中，AngularJS会负责重写`<a href=""></a>`中的链接。也就是说AngularJS会根据浏览器的能力在编译时决定是否要重写`href=""`中的链接。

例如`<a href="/person/42?all=true">Person</a>`这个标签，在老式浏览器中会被重写成标签模式的URL：`/index.html#!/person/42?all=true`。但在现代浏览器中会URL会保持本来的样子。

后端服务器也需要支持URL重写,服务器需要确保所有请求都返回index.html,以支持HTML5模式。这样才能确保由AngularJS应用来处理路由。

当在HTML5模式的AngularJS中写链接时，永远都不要使用相对路径。如果你的应用是在根路径中加载的，这不会有什么问题，但如果是在其他路径中，AngularJS应用就无法正确处理路由了。

另一个选择是在HTML文档的HEAD中用`<base>`标签来指定应用的基础URL：

```
<base href="/base/url" />
```

12.5.2 路由事件

`$route`服务在路由过程中的每个阶段都会触发不同的事件,可以为这些不同的路由事件设置监听器并做出响应。

这个功能对于控制不同的路由事件，以及探测用户的登录和授权状态等场景是非常有用的。

我们需要给路由设置事件监听器，用`$rootScope`来监听这些事件。

1. $routeChangeStart

AngularJS在路由变化之前会广播`$routeChangeStart`事件。在这一步中，路由服务会开始加载路由变化所需要的所有依赖，并且模板和`resolve`键中的promise也会被resolve。

```
angular.module('myApp', [])
    .run(['$rootScope', '$location', function($rootScope, $location)  {
        $rootScope.$on('$routeChangeStart', function(evt, next, current) {
        });
    }]);
```

`$routeChangeStart`事件带有两个参数：

- 将要导航到的下一个URL；
- 路由变化前的URL。

2. $routeChangeSuccess

AngularJS会在路由的依赖被加载后广播`$routeChangeSuccess`事件。

```
angular.module('myApp', [])
    .run(['$rootScope', '$location', function($rootScope, $location) {
        $rootScope.$on('$routeChangeSuccess', function(evt, next, previous) {
        });
    }]);
```

`$routeChangeStart`事件带有三个参数：

- 原始的AngularJS evt对象；
- 用户当前所处的路由；
- 上一个路由（如果当前是第一个路由，则为undefined）。

3. $routeChangeError

AngularJS会在任何一个promise被拒绝或者失败时广播$routeChangeError事件。

```
angular.module('myApp', [])
    .run(function($rootScope, $location) {
        $rootScope.$on('$routeChangeError', function(current, previous, rejection) {
        });
    });
```

$routeChangeError事件有三个参数：

- 当前路由的信息；
- 上一个路由的信息；
- 被拒绝的promise的错误信息。

4. $routeUpdate

AngularJS在reloadOnSearch属性被设置为false的情况下，重新使用某个控制器的实例时，会广播$routeUpdate事件。

12.5.3 关于搜索引擎索引

Web爬虫对于JavaScript的胖客户端应用无能为力。为了在应用的运行过程中给爬虫提供支持，我们需要在头部添加meta标签。这个元标记会让爬虫请求一个带有空的转义片段参数的链接，服务器根据请求返回对应的HTML代码片段。

```
<meta name="fragment" content="!"/>
```

12

12.6 更多关于路由的内容

12.6.1 页面重新加载

$location服务不会重新加载整个页面，它只会单纯地改变URL。如果我们想重新加载整个页面，需要用$window服务来设置地址。

```
$window.location.href = "/reload/page";
```

12.6.2 异步的地址变化

如果我们想要在作用域的生命周期外使用$location服务，必须用$apply函数将变化抛到应用外部。因为$location服务是基于$digest来驱动浏览器的地址变化，以使路由事件正常工作的。

第 13 章

依赖注入

一个对象通常有三种方式可以获得对其依赖的控制权：

(1) 在内部创建依赖；

(2) 通过全局变量进行引用；

(3) 在需要的地方通过参数进行传递。

依赖注入是通过第三种方式实现的。其余两种方式会带来各种问题，例如污染全局作用域，使隔离变得异常困难等。依赖注入是一种设计模式，它可以去除对依赖关系的硬编码，从而可以在运行时改变甚至移除依赖关系。

在运行时修改依赖关系的能力对测试来讲是非常理想的，因为它允许我们创建一个隔离的环境，从而在测试环境可以使用模拟的对象取代生产环境中的真实对象。

从功能上看，依赖注入会事先自动查找依赖关系，并将注入目标告知被依赖的资源，这样就可以在目标需要时立即将资源注入进去。

在编写依赖于其他对象或库的组件时，我们需要描述组件之间的依赖关系。在运行期，注入器会创建依赖的实例，并负责将它传递给依赖的消费者。

```
// 出自Angular文档的优秀示例
function SomeClass(greeter) {
    this.greeter = greeter;
}
SomeClass.prototype.greetName = function(name) {
    this.greeter.greet(name);
};
```

> 注意，示例代码在全局作用域上创建了一个控制器，这并不是一个好主意，这里只是为了方便演示。

SomeClass能够在运行时访问到内部的greeter，但它并不关心如何获得对greeter的引用。为了获得对greeter实例的引用，SomeClass的创建者会负责构造其依赖关系并传递进去。

基于以上原因，AngularJS使用$injetor（注入器服务）来管理依赖关系的查询和实例化。事实上，$injetor负责实例化AngularJS中所有的组件，包括应用的模块、指令和控制器等。

在运行时，任何模块启动时$injetor都会负责实例化，并将其需要的所有依赖传递进去。

例如下面这段代码。这是一个简单的应用，声明了一个模块和一个控制器：

```
angular.module('myApp', [])
.factory('greeter', function() {
    return {
        greet: function(msg) {alert(msg);}
    }
})
.controller('MyController',
    function($scope, greeter) {
        $scope.sayHello = function() {
            greeter.greet("Hello!");
        };
});
```

当AngularJS实例化这个模块时，会查找greeter并自然而然地把对它的引用传递进去：

```
<div ng-app="myApp">
    <div ng-controller="MyController">
        <button ng-click="sayHello()">Hello</button>
    </div>
</div>
```

而在内部，AngularJS的处理过程是下面这样的：

```
// 使用注入器加载应用
var injector = angular.injector(['ng', 'myApp']);
 // 通过注入器加载$controller服务：var $controller = injector.get('$controller');
var scope = injector.get('$rootScope').$new();
// 加载控制器并传入一个作用域，同AngularJS在运行时做的一样
var MyController = $controller('MyController', {$scope: scope})
```

上面的代码中并没有说明是如何找到greeter的，但是它的确能正常工作，因为$injector会负责为我们查找并加载它。

AngularJS通过annotate函数，在实例化时从传入的函数中把参数列表提取出来。在Chrome的开发者工具中输入下面的代码可以查看这个函数：

```
> injector.annotate(function($q, greeter) {})
["$q", "greeter"]
```

在任何一个AngularJS的应用中，都有$injector在进行工作，无论我们知道与否。当编写控制器时，如果没有使用[]标记或进行显式的声明，$injector就会尝试通过参数名推断依赖关系。

13.1　推断式注入声明

如果没有明确的声明，AngularJS会假定参数名称就是依赖的名称。因此，它会在内部调用函数对象的toString()方法，分析并提取出函数参数列表，然后通过$injector将这些参数注入进对象实例。注入的过程如下：

```
injector.invoke(function($http, greeter) {});
```

请注意，这个过程只适用于未经过压缩和混淆的代码，因为AngularJS需要原始未经压缩的参数列表来进行解析。

有了这个根据参数名称进行推断的过程，参数顺序就没有什么重要的意义了，因为AngularJS

会帮助我们把属性以正确的顺序注入进去。

> JavaScript的压缩器通常会将参数名改写成简单的字符，以减小源文件体积（同时也会删除空格、空行和注释等）。如果我们不明确地描述依赖关系，AngularJS将无法根据参数名称推断出实际的依赖关系，也就无法进行依赖注入。

13.2　显式注入声明

AngularJS提供了显式的方法来明确定义一个函数在被调用时需要用到的依赖关系。通过这种方法声明依赖，即使在源代码被压缩、参数名称发生改变的情况下依然能够正常工作。

可以通过`$inject`属性来实现显式注入声明的功能。函数对象的`$inject`属性是一个数组，数组元素的类型是字符串，它们的值就是需要被注入的服务的名称。

下面是示例代码：

```
var aControllerFactory =
function aController($scope, greeter) {
    console.log("LOADED controller", greeter);
    // ……控制器
};
aControllerFactory.$inject = ['$scope', 'greeter']; // Greeter服务
  console.log("greeter service");
}
// 我们应用的控制器
angular.module('myApp', [])
  .controller('MyController', aControllerFactory)
  .factory('greeter', greeterService);
// 获取注入器并创建一个新的作用域
var injector = angular.injector(['ng', 'myApp']),
    controller = injector.get('$controller'),
    rootScope = injector.get('$rootScope'),
    newScope = rootScope.$new();
// 调用控制器
controller('MyController', {$scope: newScope});
```

对于这种声明方式来讲，参数顺序是非常重要的，因为`$inject`数组元素的顺序必须和注入参数的顺序一一对应。这种声明方式可以在压缩后的代码中运行，因为声明的相关信息已经和函数本身绑定在一起了。

13.3　行内注入声明

AngularJS提供的注入声明的最后一种方式，是可以随时使用的行内注入声明。这种方式其实是一个语法糖，它同前面提到的通过`$inject`属性进行注入声明的原理是完全一样的，但允许我们在函数定义时从行内将参数传入。此外，它可以避免在定义过程中使用临时变量。

在定义一个AngularJS的对象时，行内声明的方式允许我们直接传入一个参数数组而不是一个函数。数组的元素是字符串，它们代表的是可以被注入到对象中的依赖的名字，最后一个参数就是依赖注入的目标函数对象本身。

示例如下：

```
angular.module('myApp')
    .controller('MyController', ['$scope', 'greeter', function($scope, greeter) {

    }]);
```

由于需要处理的是一个字符串组成的列表，行内注入声明也可以在压缩后的代码中正常运行。通常通过括号和声明数组的[]符号来使用它。

13.4 $injector API

在实际工作中，我们很少直接同$injector打交道，但是了解一下$injector有哪些API，可以帮助我们更好地理解它是如何运作的。

13.4.1 annotate()

annotate()方法的返回值是一个由服务名称组成的数组，这些服务会在实例化时被注入到目标函数中。annotate()方法可以帮助$injector判断哪些服务会在函数被调用时注入进去。

annotate()方法可以接受一个参数：

- fn（函数或数组）

参数fn可以是一个函数，也可以是一个数组。annotate()方法返回一个数组，数组元素的值是在调用时被注入到目标函数中的服务的名称。

```
var injector = angular.injector(['ng', 'myApp']);
injector.annotate(function($q, greeter) {});
// ['$q', 'greeter']
```

可以在Chrome的调试器中试试上面这段代码。

13.4.2 get()

get()方法返回一个服务的实例，可以接受一个参数：

- name（字符串）

参数name是想要获取的实例的名称。

get()根据名称返回服务的一个实例。

13.4.3 has()

has()方法返回一个布尔值，在$injector能够从自己的注册列表中找到对应的服务时返回true，否则返回false。它能接受一个参数：

- name（字符串）

参数name是我们想在注入器的注册列表中查询的服务名称。

13.4.4 instantiate()

instantiate()方法可以创建某个JavaScript类型的实例。它会通过new操作符调用构造函数，并将所有参数都传递给构造函数。它可以接受两个参数。

- Type（函数）

构造函数。

- locals（对象，可选）

这是一个可选的参数，提供了另一种传递参数的方式。

instantiate()方法返回Type的一个新实例。

13.4.5 invoke()

invoke()方法会调用方法并从$injector中添加方法参数。

invoke()方法接受三个参数。

fn(function)

这个函数就是要调用的函数。这个函数的参数由函数声明设置。

self (object-可选)

self参数允许我们设置调用方法的this参数。

locals (object-可选)

这个可选参数提供另一种方式在函数被调用时传递参数名给该函数。

invoke()方法返回fn函数返回的值。

13.5 ngMin

上面介绍了三种声明依赖注入的方式，可以在定义函数时选择任意一种合适的方式。但在实际生产过程中，当代码体积变得非常庞大时，写代码还要关心参数顺序将是一个耗费心力的工作。

通过使用ngMin这个工具，能够减少我们定义依赖关系所需的工作量。ngMin是一个为AngularJS应用设计的预压缩工具，它会遍历整个AngularJS应用并帮助我们设置好依赖注入。

例如，它会将如下代码：

```
angular.module('myApp', [])
.directive('myDirective', function($http) { })
.controller('IndexController', function($scope, $q) {
});
```

转换成下面的形式：

```
angular.module('myApp', [])
.directive('myDirective', ['$http', function ($http) { }])
.controller('IndexController', [ '$scope', '$q',function ($scope, $q) {} ]);
```

ngMin可以显著减少代码输入的工作量，并保持源文件的整洁。

13.5.1 安装

可以通过npm包管理工具来安装ngMin：

```
$ npm install -g ngmin
```

如果正在使用Grunt，我们可以安装grunt-ngmin插件。如果正在使用Rails，也可以通过Ruby的包管理工具gem来安装ngmin-rails。

13.5.2 使用ngMin

我们可以在命令行界面单独使用ngMin，可以通过标准输入输出设备或标准输出流传入input.js和output.js两个参数，例如：

```
$ ngmin input.js output.js
#或者
$ ngmin < input.js > output.js
```

input.js是源文件，而 output.js则是转换过注入声明后的输出文件。

13.5.3 工作原理

在其内部，ngMin使用抽象语法树（Abstract Syntax Tree，AST）来遍历JavaScript源代码。借助名为astral的AST工具框架的帮助，它可以将必要的声明代码添加进源文件，并用escodegen将转换后的源文件输出。

ngMin希望我们的AngularJS源代码只由逻辑定义组成。如果我们书写代码的语法和这本书里的一样，那么ngMin就可以对其进行解析和预压缩。

第 14 章

服　　务

14

到目前为止，我们只关心视图是如何同$scope绑定在一起，以及控制器是如何管理数据的。出于内存占用和性能的考虑，控制器只会在需要时被实例化，并且不再需要就会被销毁。这意味着每次切换路由或重新加载视图时，当前的控制器会被AngularJS清除掉。

服务提供了一种能在应用的整个生命周期内保持数据的方法，它能够在控制器之间进行通信，并且能保证数据的一致性。

服务是一个单例对象，在每个应用中只会被实例化一次（被$injector实例化），并且是延迟加载的（需要时才会被创建）。服务提供了把与特定功能相关联的方法集中在一起的接口。

以AngularJS的$http服务为例，它提供了对浏览器的XMLHttpRequest对象的底层访问功能，我们可以通过$http的API同XMLHttpRequest进行交互，而不需要因为调用这些底层代码而污染应用。

```
// 示例服务，在应用的整个生命周期内保存current_user
angular.module('myApp', [])
.factory('UserService', function($http) {
    var current_user;

    return {
        getCurrentUser: function() {
            return current_user;
        },
        setCurrentUser: function(user) {
            current_user = user;
        }
    };
});
```

AngularJS提供了一些内置服务，在任何地方使用它们的方式都是统一的。同时，为复杂应用创建我们自己的服务也是非常有用的。

在AngularJS中创建自己的服务是非常容易的：只需要注册这个服务即可。服务被注册后，AngularJS编译器就可以引用它，并且在运行时把它当作依赖加载进来。服务名称的注册表使得在测试中伪造和剔除相互隔离的应用依赖变得非常容易。

14.1　注册一个服务

用$injector来创建和注册服务有好几种方式，本章后面会介绍它们。

使用angular.module的factory API创建服务，是最常见也是最灵活的方式：

```
angular.module('myApp.services', [])
.factory('githubService', function() {
    var serviceInstance = {};
    // 我们的第一个服务
    return serviceInstance;
});
```

尽管githubService没做什么有趣的事情，但现在它已经用githubService作为名字注册成为这个AngularJS应用的一个服务了。

服务的工厂函数用来生成一个单例的对象或函数，这个对象或函数就是服务，它会存在于应用的整个生命周期内。当我们的AngularJS应用加载服务时，这个函数会被执行并返回一个单例的服务对象。

同创建控制器的方法一样，服务的工厂函数既可以是一个函数也可以是一个数组：

```
// 用方括号声明工厂
angular.module('myApp.services', [])
    .factory('githubService', [function($http) { }]);
```

例如，githubService需要访问$http服务，所以我们将$http服务当作AngularJS应用的一个依赖，并将它注入到工厂函数中。

```
angular.module('myApp.services', [])
.factory('githubService', function($http) {
    // 我们的serviceInstance现在可以在函数定义中访问$http服务
    var serviceInstance = {};
    return serviceInstance;
});
```

现在，无论何处需要访问GitHub API都不需要通过$http来进行了，可以通过githubService来代替，并让它处理所有复杂的业务逻辑和远程服务。

GitHub API提供了一个读取用户活动流的方法（活动流就是用户记录在GitHub中的最近的事件列表）。在我们的服务中，可以创建一个访问这个API的方法，并将API的请求结果返回。

通过将方法设置为服务对象的一个属性来将其暴露给外部。

```
angular.module('myApp.services', [])
.factory('githubService', function($http) {
    var githubUrl = 'https://api.github.com';

    var runUserRequest = function(username, path) {
        // 从使用JSONP调用Github API的$http服务中返回promise
        return $http({
            method: 'JSONP',
            url: githubUrl + '/users/' +
                  username + '/' +
                  path + '?callback=JSON_CALLBACK'
        });
    };
    // 返回带有一个events函数的服务对象
    return {
        events: function(username) {
            return runUserRequest(username, 'events');
```

14

```
        }
    };
});
```

githubService中只包含了一个方法，可以在应用的模块中调用。

14.2　使用服务

可以在控制器、指令、过滤器或另外一个服务中通过依赖声明的方式来使用服务。AngularJS会像平时一样在运行期自动处理实例化和依赖加载的相关事宜。

将服务的名字当作参数传递给控制器函数，可以将服务注入到控制器中。当服务成为了某个控制器的依赖，就可以在控制器中调用任何定义在这个服务对象上的方法。

```
angular.module('myApp', ['myApp.services'])
.controller('ServiceController', function($scope, githubService) {
    // 我们可以调用对象的事件函数
    $scope.events = githubService.events('auser');
});
```

githubService服务已经被注入到ServiceController中，可以像使用任何其他服务一样使用它。

修改一下例子，用视图中输入的GitHub用户名为参数来访问GitHub API。同第2章介绍的内容一样，将username属性和视图进行绑定。

```
<div ng-controller="ServiceController">
    <label for="username">
        Type in a GitHub username
    </label>
    <input type="text"
        ng-model="username"
        placeholder="Enter a GitHub username" />
    <ul>
        <li ng-repeat="event in events">
        <!--
            event.actor and event.repo are returned
            by the github API. To view the raw
            API, uncomment the next line:
        -->
        <!-- {{ event | json }} -->
          {{ event.actor.login }} {{ event.repo.name }}
        </li>
    </ul>
</div>
```

基于双向数据绑定，我们现在可以通过监视$scope.username来响应视图中的数据变化。

```
.controller('ServiceController',
    function($scope, githubService) {
    // 注意username属性的变化
    // 如果有变化就运行该函数
    $scope.$watch('username', function(newUsername) {
        // 从使用JSONP调用Github API的$http服务中返回promise
        githubService.events(newUsername)
            .success(function(data, status, headers) {
                // success函数在数据中封装响应
```

```
                // 因此我们需要调用data.data来获取原始数据
                $scope.events = data.data;
            })
        });
    });
```

> 由于$http返回的是promise对象，可以通过.success()方法像直接调用$http一样调用返回的对象。

> 和上面的例子一样，并不推荐在控制器中使用$watch，这里只是为了方便演示。在实际生产中会将这个功能封装进一个指令，并在指令中设置$watch。

在这个例子中，你可能会注意到在输入字段发生变化前，有一个延时。如果不延时，将导致输入字段中的任何一个键盘输入都会让终端对GitHub API进行调用，这显然不是我们希望的。

通过内置服务$timeout来介绍一下这个延时。同注入githubService一样，需要将$timeout服务注入到控制器中：

```
app.controller('ServiceController', function($scope, $timeout, githubService) {
});
```

> 在自定义服务之前注入所有的AngularJS内置服务，这是约定俗成的规则。

现在可以在控制器中使用$timeout服务了。在这个例子中$timeout服务会取消所有网络请求，并在输入字段的两次变化之间延时350 ms。换句话说，如果用户两次输入之间有350 ms的间隔，就推断用户已经完成了输入，然后开始向GitHub发送请求：

```
app.controller('ServiceController', function($scope, $timeout, githubService) {
        // 和上面的示例一样，添加了$timeout服务
        var timeout;
        $scope.$watch('username', function(newUserName) {
            if (newUserName) {
                // 如果在进度中有一个超时(timeout)
                if (timeout) $timeout.cancel(timeout);
                timeout = $timeout(function() {
                    githubService.events(newUserName)
                        .success(function(data, status) {
                            $scope.events = data.data;
                        });
                }, 350);
            }
        });
    });
```

到现在为止，我们只介绍了服务如何将类似的功能打包在一起，而使用服务也是在控制器之间共享数据的典型方法。

例如，如果我们的应用需要后端服务的授权，可以创建一个SessionsService服务处理用户的授权过程，并保存服务端返回的令牌。当应用中任何地方要发送一个需要授权的请求，可以通过SessionsService来访问令牌。

如果我们的应用中有一个用来设置GitHub用户名的设置页面，我们希望在应用中所有的控制器之间共享用户名。

为了在控制器之间共享数据，需要在服务中添加一个用来储存用户名的方法。记住，服务在应用的生命周期内是单例模式的，因此可以将用户名安全地储存在其中。

```
angular.module('myApp.services', [])
    .factory('githubService', function($http) {
        var githubUrl = 'https://api.github.com',
            githubUsername;

        var runUserRequest = function(path) {
            // 从使用JSONP的Github API的$http服务中返回promise
            return $http({
                method: 'JSONP',
                url: githubUrl + '/users/' +
                    githubUsername + '/' +
                    path + '?callback=JSON_CALLBACK'
            });
        };
        // 返回带有两个方法的服务对象
        // 事件
        // 和setUsername
        return {
            events: function() {
                return runUserRequest('events');
            },
            setUsername: function(username) {
                githubUsername = username;
            }
        };
    });
```

现在，服务中有一个setUsername方法，用来保存当前的GitHub用户名了。

githubService可以注入到应用的任何一个控制器中，并可以在控制器中调用events()方法，且无须担心当前作用域对象上的用户名是否是正确的。

```
angular.module('myApp', ['myApp.services'])
.controller('ServiceController',
function($scope, githubService) {
    $scope.setUsername =
        githubService.setUsername;
});
```

14.3　创建服务时的设置项

在AngularJS应用中，factory()方法是用来注册服务的最常规方式，同时还有其他一些API可以在特定情况下帮助我们减少代码量。

共有5种方法用来创建服务：

- factory()
- service()
- constant()
- value()
- provider()

14.3.1 factory()

如前所见，factory()方法是创建和配置服务的最快捷方式。factory()函数可以接受两个参数。

- name（字符串）

需要注册的服务名。

- getFn（函数）

这个函数会在AngularJS创建服务实例时被调用。

```
angular.module('myApp')
.factory('myService', function() {
    return {
        'username': 'auser'
    };
});
```

因为服务是单例对象，getFn在应用的生命周期内只会被调用一次。同其他AngularJS的服务一样，在定义服务时，getFn可以接受一个包含可被注入对象的数组或函数。

getFn函数可以返回简单类型、函数乃至对象等任意类型的数据（同value()函数类似）。

```
angular.module('myApp')
.factory('githubService', ['$http', function($http) {
    return {
        getUserEvents: function(username) {
            // ...
        }
    };
}]);
```

14.3.2 service()

使用service()可以注册一个支持构造函数的服务，它允许我们为服务对象注册一个构造函数。

service()方法接受两个参数。

- name（字符串）

要注册的服务名称。

- constructor（函数）

构造函数，我们调用它来实例化服务对象。

service()函数会在创建实例时通过new关键字来实例化服务对象。

```
var Person = function($http) {
    this.getName = function() {
        return $http({ method: 'GET', url: '/api/user'});
     };
};
angular.service('personService', Person);
```

14

14.3.3 provider()

所有服务工厂都是由$provide服务创建的，$provide服务负责在运行时初始化这些提供者。

提供者是一个具有$get()方法的对象，$injector通过调用$get方法创建服务实例。$provider提供了数个不同的API用于创建服务，每个方法都有各自的特殊用途。

所有创建服务的方法都构建在provider方法之上。provider()方法负责在$providerCache中注册服务。

从技术上说，当我们假定传入的函数就是$get()时，factory()函数就是用provider()方法注册服务的简略形式。

下面两种方法的作用完全一样，并且会创建同一个服务。

```
angular.module('myApp')
.factory('myService', function() {
    return {
        'username': 'auser'
    };
})
// 这与上面工厂的用法等价
.provider('myService', {
    $get: function() {
        return {
            'username': 'auser'
        };
    }
});
```

是否可以一直使用.factory()方法来代替.provider()呢？

答案取决于是否需要用AngularJS的.config()函数来对.provider()方法返回的服务进行额外的扩展配置。同其他创建服务的方法不同，config()方法可以被注入特殊的参数。

比如我们希望在应用启动前配置githubService的URL：

```
// 使用`.provider`注册该服务
angular.module('myApp', [])
.provider('githubService', function($http) {
    // 默认的，私有状态
    var githubUrl = 'https://github.com'

    setGithubUrl: function(url) {
        // 通过.config改变默认属性
        if (url) { githubUrl = url }
    },
    method: JSONP, // 如果需要，可以重写

    $get: function($http) {
        self = this;
        return $http({ method: self.method, url: githubUrl + '/events'});
    }
});
```

通过使用.provider()方法，可以在多个应用使用同一个服务时获得更强的扩展性，特别是在不同应用或开源社区之间共享服务时。

在上面的例子中，provider()方法在文本githubService后添加Provider生成了一个新的提供者，githubServiceProvider可以被注入到config()函数中。

```
angular.module('myApp', [])
.config(function(githubServiceProvider) {
    githubServiceProvider.setGithubUrl("git@github.com");
});
```

如果希望在config()函数中可以对服务进行配置，必须用provider()来定义服务。

provider()方法为服务注册提供者。可以接受两个参数。

- name（字符串）

name参数在providerCache中是注册的名字。name+Provider会成为服务的提供者。同时name也是服务实例的名字。

例如，如果定义了一个githubService，那它的提供者就是githubServiceProvider。

- aProvider（对象/函数/数组）

aProvider可以是多种形式。

如果aProvider是函数，那么它会通过依赖注入被调用，并且负责通过$get方法返回一个对象。

如果aProvider是数组，会被当做一个带有行内依赖注入声明的函数来处理。数组的最后一个元素应该是函数，可以返回一个带有$get方法的对象。

如果aProvider是对象，它应该带有$get方法。

provider()函数返回一个已经注册的提供者实例。

直接使用provider() API是最原始的创建服务的方法：

```
// 在模块对象上直接创建provider的例子
angular.module('myApp', [])
.provider('UserService', {
    favoriteColor: null,
    setFavoriteColor: function(newColor) {
        this.favoriteColor = newColor;
    },
    // $get函数可以接受injectables
    $get: function($http) {
        return {
            'name': 'Ari',
            getFavoriteColor: function() {
                return this.favoriteColor || 'unknown';
            }
        };
     }
});
```

用这个方法创建服务，必须返回一个定义有$get()函数的对象，否则会导致错误。

可以通过注入器来实例化服务（由于AngularJS会处理服务的实例化，我们不需要自己动手，更多信息请查看第24章）：

```
// Get the injector
var injector = angular.injector(['myApp']); // Invoke our service
injector.invoke(
     ['UserService', function(UserService) {
        // UserService returns
        // {
        // 'name': 'Ari',
        //  getFavoriteColor: function() {}
        // }
        }]);
```

.provider()是非常强大的，可以让我们在不同的应用中共享服务。

了解constant()和value()方法对于创建服务也是非常重要的。

14.3.4　constant()

可以将一个已经存在的变量值注册为服务，并将其注入到应用的其他部分当中。例如，假设我们需要给后端服务一个apiKey，可以用constant()将其当作常量保存下来。

constant()函数可以接受两个参数。

- name（字符串）

需要注册的常量的名字。

- value（常量）

需要注册的常量的值（值或者对象）。

constant()方法返回一个注册后的服务实例。

```
angular.module('myApp') .constant('apiKey','123123123')
```

这个常量服务可以像其他服务一样被注入到配置函数中：

```
angular.module('myApp')
.controller('MyController', function($scope, apiKey) {
        // 可以像上面一样用apiKey作为常量
        // 用123123123作为字符串的值
        $scope.apiKey = apiKey;
});
```

这个常量不能被装饰器拦截。

14.3.5　value()

如果服务的$get方法返回的是一个常量，那就没要必要定义一个包含复杂功能的完整服务，可以通过value()函数方便地注册服务。

value()方法可以接受两个参数。

- name（字符串）

同样是需要注册的服务名。

❑ value（值）

将这个值将作为可以注入的实例返回。

value()方法返回以name参数的值为名称的注册后的服务实例。

```
angular.module('myApp')
.value('apiKey','123123123');
```

14.3.6 何时使用value()和constant()

value()方法和constant()方法之间最主要的区别是，常量可以注入到配置函数中，而值不行。

通常情况下，可以通过value()来注册服务对象或函数，用constant()来配置数据。

```
angular.module('myApp', [])
.constant('apiKey', '123123123')
.config(function(apiKey) {
    // 在这里apiKey将被赋值为123123123
    // 就像上面设置的那样
})
.value('FBid','231231231')
.config(function(FBid) {
    // 这将抛出一个错误，未知的provider: FBid
    // 因为在config函数内部无法访问这个值
});
```

14.3.7 decorator()

$provide服务提供了在服务实例创建时对其进行拦截的功能，可以对服务进行扩展，或者用另外的内容完全代替它。

装饰器是非常强大的，它不仅可以应用在我们自己的服务上，也可以对AngularJS的核心服务进行拦截、中断甚至替换功能的操作。事实上AngularJS中很多功能的测试就是借助$provide.decorator()建立的。

对服务进行装饰的场景有很多，比如对服务进行扩展，将外部数据缓存进localStorage的功能，或者对服务进行封装以便在开发中进行调试和跟踪等。

例如，我们想给之前定义的githubService服务加入日志功能，可以借助decorator()函数方便地实现这个功能，而不需要对原始的服务进行修改。

decorator()函数可以接受两个参数。

❑ name（字符串）

将要拦截的服务名称。

❑ decoratorFn（函数）

在服务实例化时调用该函数，这个函数由injector.invoke调用，可以将服务注入这个函数中。

$delegate是可以进行装饰的最原始的服务，为了装饰其他服务，需要将其注入进装饰器。

例如，下面的代码展示了如何给githubService添加装饰器，从而为每个请求都加上一个时间戳：

```
var githubDecorator = function($delegate,$log) {
    var events = function(path) {
        var startedAt = new Date();
        var events = $delegate.events(path);
        // 事件是一个promise events.finally(function() {
            $log.info("Fetching events" +
                " took " +
                (new Date() - startedAt) + "ms");
         });
        return events;
    };

    return {
        events: events
    };
};

angular.module('myApp')
.config(function($provide) {
    $provide.decorator('githubService',githubDecorator);
});
```

第 15 章

同外界通信：XHR和服务器通信

15

AngularJS应用是完全运行在客户端的应用。我们已经看到，可以构建一个完全不依赖任何后端，同时也能实现动态内容和响应的Web应用。

没有后端的支持，我们只能展示随页面一起加载进来的数据。AngularJS提供了几种方式将应用同来自远程服务器的信息集成在一起。

15.1 使用$http

我们可以使用内置的$http服务直接同外部进行通信。$http服务只是简单的封装了浏览器原生的XMLHttpRequest对象。

$http服务是只能接受一个参数的函数，这个参数是一个对象，包含了用来生成HTTP请求的配置内容。这个函数返回一个promise对象，具有success和error两个方法。

查看15.2节来了解关于可选配置选项的详细信息。

这两个方法最基本的使用场景如下：

```
$http({
  method: 'GET',
  url: '/api/users.json'
}).success(function(data,status,headers,config) {
  // 当相应准备就绪时调用
}).error(function(data,status,headers,config) {
  // 当响应以错误状态返回时调用
});
```

请注意，看上去我们向$http中传入了一个回调函数供响应返回时调用，但事实并非如此。这个方法实际上返回了一个promise对象。

当promise返回时，我们可以将$http方法的运行结果当作变量一并返回，并将其他promise同它串联在一起，进行链式的调用。

在创建服务时会频繁使用链式调用技术，因此服务可以返回一个promise对象，而不需要回调函数。

```
var promise = $http({
    method: 'GET',
    url: '/api/users.json'
});
```

由于$http方法返回一个promise对象，我们可以在响应返回时用then方法来处理回调。如果使用then方法，会得到一个特殊的参数，它代表了相应对象的成功或失败信息，还可以接受两个可选的函数作为参数。或者可以使用success和error回调代替。

```
promise.then(function(resp){
    // resp是一个响应对象
}, function(resp) {
    // 带有错误信息的resp
});
// 或者使用success/error方法
promise.success(function(data, status, headers, config){
    // 处理成功的响应
});
// 错误处理
promise.error(function(data, status, headers, config){
    // 处理非成功的响应
});
```

如果响应状态码在200和299之间，会认为响应是成功的，success回调会被调用，否则error回调会被调用。

> 注意，如果响应结果是重定向，XMLHttpRequest会跟进这个重定向，error回调并不会被调用。

我们可以调用HttpPromise对象上的then()、success()和error()方法。then()方法与其他两种方法的主要区别是，它会接收到完整的响应对象，而success()和error()则会对响应对象进行析构。

调用http方法后，在下一个$digest循环运行之前它并不会被真正执行。尽管大部分情况下我们都是在$apply代码快内部使用$http，但也可以在AngularJS的$digest循环以外执行这个方法。

如果要在AngularJS的$digest循环以外执行$http函数，需要将其封装在一个$apply代码快中。这样会强制digest循环执行，我们的promise可以按预期那样被resolve。

```
$scope.$apply(function(){
    $http({
        method: 'GET',
        url: '/api/users.json'
    });
});
```

快捷方法

$http服务提供了一些顺手的快捷方法供我们使用，这些方法简化了复杂设置，只需要提供URL和HTTP方法（或者POST或PUT请求中包含的数据）即可。

用这些快捷方法，可以将上面$http的GET请求修改成：

```
// 快捷的GET请求
$http.get('/api/users.json');
```

1. get()

这个方法是发送GET请求的快捷方式。

get()函数可以接受两个参数。

- url（字符串）

一个绝对或相对路径的URL，代表请求的目的地。

- config（可选，对象）

这是一个可选的设置对象。

get()方法返回HttpPromise对象。

2. delete()

这是用来发送DELETE请求的快捷方式。

delete()函数可以接受两个参数。

- url（字符串）

一个绝对或相对路径的URL，代表请求的目的地。

- config（可选，对象）

这是一个可选的设置对象。

delete()方法返回HttpPromise对象。

3. head()

这是用来发送HEAD请求的快捷方式。

head()函数可以接受两个参数。

- url（字符串）

一个绝对或相对路径的URL，代表请求的目的地。

- config（可选，对象）

这是一个可选的设置对象。

head()方法返回HttpPromise对象。

4. jsonp()

这是用来发送JSONP请求的快捷方式。

jsonp()函数可以接受两个参数。

- url（字符串）

一个绝对或相对路径的URL，代表请求的目的地。为了发送JSONP请求，其中必须包含

JSON_CALLBACK字样。例如：

```
$http
.jsonp("/api/users.json?callback=JSON_CALLBACK");
```

- config（可选，对象）

这是一个可选的设置对象。

jsonp()方法返回HttpPromise对象。

5. post()

这是用来发送POST请求的快捷方式。

post()函数可以接受三个参数。

- url（字符串）

一个绝对或相对路径的URL，代表请求的目的地。

- data（对象或字符串）

这个对象包含请求的数据。

- config（可选，对象）

这是一个可选的设置对象。

post()方法返回HttpPromise对象。

6. put()

这是用来发送PUT请求的快捷方式。

put()函数可以接受三个参数。

- url（字符串）

一个绝对或相对路径的URL，代表请求的目的地。

- data（对象或字符串）

这个对象包含请求的数据。

- config（可选，对象）

这是一个可选的设置对象。

put()方法返回HttpPromise对象。

15.2　设置对象

当我们将$http当作函数来调用时，需要传入一个设置对象，用来说明如何构造XHR对象。例如，可以像下面这样将$http当作函数来调用：

```
$http({
```

```
    method: 'GET',
    url: '/api/users.json',
    params: {
        'username': 'auser'
    }
});
```

设置对象可以包含以下键。

1. method（字符串）

这个键是我们希望发送的请求的HTTP方法。它的值是下列各项其中之一：'GET'、'DELETE'、'HEAD'、'JSONP'、'POST'、'PUT'。

2. url（字符串）

绝对或相对的URL，是请求的目标。

3. params（字符串map或对象）

这个键的值是一个字符串map或对象，会被转换成查询字符串追加在URL后面。如果值不是字符串，会被JSON序列化。

```
// 参数会转化为?name=ari的形式
$http({
    params: {'name': 'ari'}
})
```

4. data（字符串或对象）

这个对象中包含了将会被当作消息体发送给服务器的数据。通常在发送POST请求时使用。

从AngularJS 1.3开始，它还可以在POST请求里发送二进制数据。要发送一个blob对象，你可以简单地通过使用data参数来传递它。例如：

```
var blob = new Blob(['Hello World'], {type: 'text/plain'});
$http({
  method: 'POST',
  url: '/',
  data: blob
});
```

5. headers（对象）

一个列表，每一个元素都是一个函数，它会返回代表随请求发送的HTTP头。如果函数的返回值是null，对应的头不会被发送。

6. xsrfHeaderName（字符串）

保存XSFR令牌的HTTP头的名称。

7. xsrfCookieName（字符串）

保存XSFR令牌的cookie的名称。

8. transformRequest（函数或函数数组）

这是一个函数或函数数组，用来对HTTP请求的请求体和头信息进行转换，并返回转换后的

15

版本。通常用于在请求发送给服务器之前对其进行序列化。

这个函数看起来是这样的：

```
function(data,headers) {}
```

9. `transformResponse`（函数或函数数组）

这是一个函数或函数数组，用来对HTTP响应的响应体和头信息进行转换，并返回转换后的版本。通常用来进行反序列化。

这个函数看起来是这样的：

```
function(data,headers) {}
```

10. `cache`（布尔型或缓存对象）

如果`cache`属性被设置为`true`，那么AngularJS会用默认的`$http`缓存来对GET请求进行缓存。如果`cache`属性被设置为一个`$cacheFactory`对象的实例，那么这个对象会被用来对GET请求进行缓存。

11. `timeout`（数值型或promise对象）

如果`timeout`被设置为一个数值，那么请求将会在推迟`timeout`指定的毫秒数后再发送。如果被设置为一个promise对象，那么当该promise对象被resolve时请求会被中止。

12. `withCredentials`（布尔型）

如果该属性被设置为`true`，那么XHR请求对象中会设置`withCredentials`标记。

默认情况下，CORS请求不会发送cookie，而`withCredentials`标记会在请求中加入Access-Control-Allow-Credentials头，这样请求就会将目标域的cookie包含在请求中。

13. `responseType`（字符串）

`responseType`选项会在请求中设置`XMLHttpRequestResponseType`属性。我们可以使用以下HTTP请求类型其中之一：

- `""`（字符串，默认）；
- `"arraybuffer"`（`ArrayBuffer`）；
- `"blob"`（`blob`对象）；
- `"document"`（HTTP文档）；
- `"json"`（从JSON对象解析而来的JSON字符串）；
- `"text"`（字符串）；
- `"moz-blob"`（Firefox的接收进度事件）；
- `"moz-chunked-text"`（文本流）；
- `"moz-chunked-arraybuffer"`（`ArrayBuffer`流）。

15.3　响应对象

AngularJS传递给`then()`方法的响应对象包含四个属性。

- data（字符串或对象）

这个数据代表转换过后的响应体（如果定义了转换的话）。

- status（数值型）

响应的HTTP状态码。

- headers（函数）

这个函数是头信息的getter函数，可以接受一个参数，用来获取对应名字的值。例如，用如下代码获取X-Auth-ID的值：

```
$http({
    method: 'GET',
    url: '/api/users.json'
}).then (resp) {
     // 读取X-Auth-ID
     resp.headers('X-Auth-ID');
});
```

- config（对象）

这个对象是用来生成原始请求的完整设置对象。

- statusText（字符串）

这个字符串是响应的HTTP状态文本。

15.4 缓存 HTTP 请求

默认情况下，$http服务不会对请求进行本地缓存。在发送单独的请求时，我们可以通过向$http请求传入一个布尔值或者一个缓存实例来启用缓存。

```
$http.get('/api/users.json',{ cache: true })
.success(function(data) {})
.error(function(data) {});
```

第一次发送请求时，$http服务会向/api/users.json发送一个GET请求。第二次发送同一个GET请求时，$http服务会从缓存中取回请求的结果，而不会真的发送一个HTTP GET请求。

在这个例子里，由于设置了启用缓存，AngularJS默认会使用$cacheFactory,这个服务是AngularJS在启动时自动创建的。

> 更多关于使用AngularJS缓存的内容，请查看第28章。

如果想要对AngularJS使用的缓存进行更多的自定义控制，可以向请求传入一个自定义的缓存实例代替true。

例如，如果要使用LRU（Least Recenlty Used，最近最少使用）缓存，可以像下面这样传入缓存实例对象：

```
var lru = $cacheFactory('lru',{
```

```
    capacity: 20
});
// $http请求
$http.get('/api/users.json', { cache: lru })
.success(function(data){})
.error(function(data){});
```

现在，最新的20个请求会被缓存。第21个请求会导致LRU从缓存中将时间比较老的请求移除掉。

每次发送请求时都传入一个自定义缓存是很麻烦的事情（即使是在服务中）。可以通过应用的`.config()`函数给所有`$http`请求设置一个默认的缓存：

```
angular.module('myApp', [])
.config(function($httpProvider, $cacheFactory) {
    $httpProvider.defaults.cache = $cacheFactory('lru', {
        capacity: 20
    });
});
```

现在，所有的请求都会使用我们自定义的LRU缓存了。

15.5　拦截器

任何时候如果我们想要为请求添加全局功能，例如身份验证、错误处理等，在请求发送给服务器之前或者从服务器返回时对其进行拦截，是比较好的实现手段。

例如对于身份验证，如果服务器返回401状态码，我们会希望将用户重定向到登录页面。

AngularJS通过拦截器提供了一个从全局层面对响应进行处理的途径。

拦截器，尽管名字听起来很唬人，实际上是`$http`服务的基础中间件，用来向应用的业务流程中注入新的逻辑。

拦截器的核心是服务工厂（查看第14章获得更多关于服务的信息），通过向`$httpProvider.interceptors`数组中添加服务工厂，在`$httpProvider`中进行注册。

一共有四种拦截器，两种成功拦截器，两种失败拦截器。

- `request`

AngularJS通过`$http`设置对象来对请求拦截器进行调用。它可以对设置对象进行修改，或者创建一个新的设置对象，它需要返回一个更新过的设置对象，或者一个可以返回新的设置对象的promise。

- `response`

AngularJS通过`$http`设置对象来对响应拦截器进行调用。它可以对响应进行修改，或者创建一个新的响应，它需要返回一个更新过的响应，或者一个可以返回新响应的promise。

- `requestError`

AngularJS会在上一个请求拦截器抛出错误，或者promise被reject时调用此拦截器。

- `responseError`

AngularJS会在上一个响应拦截器抛出错误，或者promise被reject时调用此拦截器。

调用模块的.factory()方法来创建拦截器，可以在服务中添加一种或多种拦截器：

```
angular.module('myApp', [])
.factory('myInterceptor', function($q) {
    var interceptor = {
        'request': function(config) {
            // 成功的请求方法
            return config; // 或者 $q.when(config);
        },
        'response': function(response) {
            // 响应成功
            return response; // 或者 $q.when(config);
        },
        'requestError': function(rejection) {
            // 请求发生了错误，如果能从错误中恢复，可以返回一个新的请求或promise
            return response; // 或新的promise
            // 或者，可以通过返回一个rejection来阻止下一步
            // return $q.reject(rejection);
        },
        'responseError': function(rejection) {
            // 请求发生了错误，如果能从错误中恢复，可以返回一个新的响应或promise
            return rejection; // 或新的promise
            // 或者，可以通过返回一个rejection来阻止下一步
            // return $q.reject(rejection);
        }
    };
    return interceptor;
});
```

我们需要使用$httpProvider在.config()函数中注册拦截器：

```
angular.module('myApp', [])
.config(function($httpProvider) {
    $httpProvider.interceptors.push('myInterceptor');
});
```

15.6　设置$httpProvider

使用.config()可以向所有请求中添加特定的HTTP头，这非常有用，尤其是我们希望将身份验证的头同请求一同发送，或设置响应类型的时候。

默认的请求头保存在$httpProvider.defaults.headers.common对象中。默认的头如下所示：

```
Accept: application/json, text/plain, */*
```

通过.config()函数可以对这些头进行修改或扩充，如下所示：

```
angular.module('myApp', [])
.config(function($httpProvider) {
    $httpProvider.defaults.headers
        .common['X-Requested-By'] = 'MyAngularApp';
});
```

也可以在运行时通过$http对象的defaults属性对这些默认值进行修改。例如，通过如下方法可以动态添加一个头：

```
$http.defaults
    .common['X-Auth'] = 'RandomString';、
```

> 这个功能可以通过使用请求转换器实现，对于单个请求，也可以通过设置$http请求的headers选项实现。

也可以只对POST和PUT类型的请求进行设置。POST请求的默认头如下所示：

```
Content-Type: application/json
```

可以在.config()函数中对POST请求的头进行修改或扩充，如下所示：

```
angular.module('myApp', [])
.config(function($httpProvider) {
    $httpProvider.defaults.headers
        .post['X-Posted-By'] = 'MyAngularApp';
});
```

也可以对所有的PUT请求做同样的设置。PUT请求的默认头如下所示：

```
Content-Type: application/json
```

可以在.config()函数中对PUT请求的头进行修改或扩充，如下所示：

```
angular.module('myApp', []) .config(function($httpProvider){
    $httpProvider.defaults.headers
        .put['X-Posted-By'] = 'MyAngularApp';
});
```

15.7　使用$resource

AngularJS还提供另外一个非常有用的可选服务$resource供我们使用。这个服务可以创建一个资源对象，我们可以用它非常方便地同支持RESTful的服务端数据源进行交互，当同支持RESTful的数据模型一起工作时，它就派上用场了。

> REST是Representational State Transfer（表征状态转移）的缩写，是服务器用来智能化地提供数据服务的一种方式。更多关于REST的信息可以查看维基百科[①]。

$resource服务难以置信地方便，它对很多复杂的细节进行了抽象，只留下同后端服务器进行真正有意义的交互，前提是服务器支持RESTful的数据模型。

$resource服务可以将$http请求转换成save和update等简单形式。我们可以通过$resource服务来处理复杂的细节，而无需自己编写重复和繁琐的业务代码。

15.8　安装

ngResource模块是一个可选的AngularJS模块，支持与RESTful的后端数据源进行交互。由于ngResource模块没有默认内置在AngularJS中，因此我们需要安装并在应用中引用它。

首先需要从code.angularjs.org[②]上下载它，然后放到一个类似于js/vendor/angular-resource.js这

① http://en.wikipedia.org/wiki/Representational_State_Transfer

② http://code.angularjs.org

样的地方，使其可以在HTML页面中被引用。

同样也可以使用Bower来进行安装，这样它会被安装到Bower的组件目录中。更多关于Bower的信息，请查看34.6节。

```
$ bower install --save angular-resouce
```

这个模块需要在AngularJS之后进行引用。

```
<script src="js/vendor/angular.js"></script>
<script src="js/vendor/angular-resource.js"></script>
```

最后，需要在我们的应用中将ngResource当作依赖进行引用：

```
angular.module('myApp', ['ngResource']);
```

现在可以使用$resource服务了。

15.9　应用$resource

$resource服务本身是一个创建资源对象的工厂。返回的$resource对象中包含了同后端服务器进行交互的高层API。

```
var User = $resource('/api/users/:userId.json',
    {
    }userId: '@id'
});
```

$resource返回包含了几个默认动作的资源类对象。可以把User对象理解成同RESTful的后端服务进行交互的接口。

资源类对象本身包含的方法可以同后端服务进行简洁的交互。

默认情况下，这个对象包含了五个常用的方法，可以同资源集合进行交互，或者创建资源对象的实例。它会生成两个基于HTTP GET类型的方法，以及三个非GET类型的方法。

15.9.1　基于HTTP GET方法

两个HTTP GET类型的方法可以接受下面三个参数。

- params（对象）

随请求一起发送的参数。它们可以是URL中的具名参数，也可以是查询参数。

- successFn（函数）

当HTTP响应成功时的回调函数。

- errorFn（函数）

当HTTP响应非成功时的回调函数。

1. **get(params, successFn, errorFn)**

get方法向指定URL发送一个GET请求，并期望一个JSON类型的响应。

15

像上面那样不定义具体的参数，get()请求通常被用来获取单个资源。

```
// 发起一个请求:
// GET /api/users
User.get(function(resp) {
    // 处理响应成功
}, function(err) {
    // 处理错误
});
```

如果参数中传入了具名参数（我们例子中的参数是id），那么get()方法会向包含id的URL发送请求：

```
// 发起一个请求:
// GET /api/users/123
User.get({
    id: '123'
}, function(resp) {
    // 处理响应成功
}, function(err) {
    // 处理错误
});
```

2. query(params, successFn, errorFn)

query向指定URL发送一个GET请求，并期望返回一个JSON格式的资源对象集合。

```
// 发起一个请求:
// GET /api/users
User.query(function(users) {
    // 读取集合中第一个用户
    var user = users[0];
});
```

query()和get()方法之间唯一的区别是AngularJS期望query()方法返回数组。

15.9.2　基于非HTTP GET类型的方法

三个基于非HTTP GET类型的方法可以接受下面四个参数。

- params（对象）

随请求一起发送的参数。它们可以是URL中的具名参数，也可以是查询参数。

- postData（对象）

这个对象是随请求发送的数据体。

- successFn（函数）

当HTTP响应成功时的回调函数。

- errorFn（函数）

当HTTP响应非成功时的回调函数。

1. save(params, payload, successFn, errorFn)

save方法向指定URL发送一个POST请求，并用数据体来生成请求体。save()方法用来在服

务器上生成一个新的资源。

```
// 发起一个请求：
// POST /api/users
// with the body {name: 'Ari'}
User.save({}, {
    name: 'Ari'
}, function(response) {
    // 处理响应成功
}, function(response) {
    // 处理非成功响应
});
```

2. `delete(params, payload, successFn, errorFn)`

`delete`方法会向指定URL发送一个DELETE请求，并用数据体来生成请求体。它被用来在服务器上删除一个实例：

```
// 发起一个请求：
// DELETE /api/users
User.delete({}, {
    id: '123'
}, function(response) {
    // 处理成功的删除响应
}, function(response) {
    // 处理非成功的删除响应
});
```

3. `remove(params, payload, successFn, errorFn)`

`remove`方法和`delete()`方法的作用是完全相同的，它存在的意义是因为`delete`是JavaScript的保留字，在IE浏览器中会导致额外的问题。

```
// 发起一个请求：
// DELETE /api/users
User.remove({}, {
    id: '123'
}, function(response) {
    // 处理成功的删除响应
}, function(response) {
    // 处理非成功的删除响应
});
```

15.9.3　`$resource`实例

上述方法返回数据时，响应会被一个原型类所包装，并在实例上添加一些有用的方法。

实例对象上会被添加下面三个实例方法：

- `$save()`
- `$remove()`
- `$delete()`

除非在一个单独的资源上而不是一个集合上被调用，否则这三个方法与资源上对应的方法是一样的。

15

这三个方法可以在资源实例上被调用。如下所示：

```
// 使用实例方法$save()
User.get({id: '123'}, function(user) {
    user.name = 'Ari';
    user.$save(); // Save the user
});
// This is equivalent to the collection-level
// resource call
User.save({id: '123'}, {name: 'Ari'});
```

15.9.4　$resource实例是异步的

需要格外注意，这三个方法在调用时$resource对象会立即返回一个空的数据引用。由于所有方法都是异步执行的，所以这个数据是一个空的引用，并不是真实的数据。

因此，虽然获取实例的调用看起来是同步的，但实际上不是。事实上，它只是数据的引用，当数据从服务器返回后AngularJS会自动将数据填充进去。

```
// $scope.user将为空
$scope.user = User.get({id: '123'});
```

这些方法也提供了回调函数，在数据返回时按预期的方式调用：

```
User.get({id: '123'}, function(user) {
    $scope.user = user;
});
```

15.9.5　附加属性

$resource集合和实例有两个特殊的属性用来同底层的数据定义进行交互。

- $promise（promise）

$promise属性是为$resource生成的原始promise对象。这个属性是特别用来同$routeProvider.when()在resolve时进行连接的。

如果请求成功了，资源实例或集合对象会随promise的resolve一起返回。如果请求失败了，promise被resolve时会返回HTTP响应对象，其中没有resource属性。

- $resolved（布尔型）

$resolved属性在服务器首次响应时会被设置为true（无论请求是否成功）。

15.10　自定义$resource 方法

尽管$resource服务提供了五种方法供我们使用，但它本身也具有良好扩展性，我们可以用自定义方法对资源对象进行扩展。

为了在$resource对象中创建自定义方法，需要向包含修改过的$http设置对象的资源类传入第三个参数，它被当作自定义方法。

在这个对象中，键是方法的名称，值是$http设置对象。

```
var User = $resource('/api/users/:userId.json', {
    userId: '@id'
    sendEmail: {
        method: 'POST'
    },
    allInboxes: {
        method: 'JSONP',
        isArray: true
    }
});
```

借助这个User资源，资源集合（User资源对象）中的个体实例现在可以使用sendEmail()和update()方法了（也就是user.$sendEmail()和user.$update()）。

15.11　$resource 设置对象

$resource设置对象和$http设置对象十分相似，仅有少量的不同。

对象中的值，也就是动作，是资源对象中某个方法的名字。

它可以包含以下键。

1. method（字符串）

method指的是我们想要用来发送HTTP请求的方法。它必须是以下值之一：'GET'、'DELETE'、'JSONP'、'POST'、'PUT'。

2. url（字符串）

一个URL，用来覆盖为该方法的具体路由设置的URL。

3. params（字符串map或对象）

这个键中包含了此动作可选的预绑定参数。如果任何一个值都是函数，那么每当我们需要读取一个请求的参数时，它就会被执行一次。

4. isArray（布尔型）

如果isArray被设置为true，那么这个动作返回的对象会以数组的形式返回。

5. transformRequest（函数或函数数组）

这个函数或函数数组用来对HTTP请求的请求体和头信息进行转换，并返回转换后的版本。通常用来进行序列化。

```
var User = $resource('/api/users/:id',{
    id: '@'
}, {
    sendEmail: {
        method: 'PUT',
        transformRequest: function(data, headerFn) {
            // 返回修改后的请求数据
            return JSON.stringify(data);
        }
```

15

```
    }
});
```

6. transformResponse（函数或函数数组）

这个函数或函数数组用来对HTTP响应体和头信息进行转换，并返回转换后的版本。通常用来进行反序列化。

```
var User = $resource('/api/users/:id',{
    id:'@'
}, {
    sendEmail: {
        method: 'PUT',
        transformResponse: function(data, headerFn)
            {
            // Return modified data for the response
            return JSON.parse(data);
        }
    }
});
```

7. cache（布尔型或缓存对象）

如果cache属性被设置为true，那么AngularJS会用默认的$http缓存对GET请求进行缓存。如果cache属性被设置为$cacheFactory对象的一个实例，那么这个对象会用来对GET请求进行缓存。

如果cache属性被设置为false，那么$resource服务所发送的请求不会被缓存。

8. timeout（数值型或promise对象）

如果timeout被设置为一个数值，那么请求将会在推迟timeout指定的毫秒数后再发送。如果被设置为一个promise对象，那么当该promise对象被resolve时，请求会被中止。

9. withCredentials（布尔型）

如果该属性被设置为true，那么XHR请求对象中会设置withCredentials标记。

默认情况下，CORS请求不会发送cookie，而withCredentials标记会在请求中加入Access-Control-Allow-Credentials头，这样请求就会携带目标域的cookie。

10. responseType（字符串）

responseType选项会在请求中设置XMLHttpRequestResponseType属性。我们可以使用以下HTTP请求类型之一：

- " "（字符串，默认）；
- "arraybuffer"（ArrayBuffer）；
- "blob"（blob对象）；
- "document"（HTTP文档）；
- "json"（从JSON对象解析而来的JSON字符串）；
- "text"（字符串）；
- "moz-blob"（Firefox的接收进度事件）；
- "moz-chunked-text"（文本流）；

- "moz-chunked-arraybuffer"（ArrayBuffer流）。

11. **interceptor**（对象）

拦截器属性有两个可选的方法：response或responseError。这些拦截器像普通的$http拦截器一样，由$http请求对象调用。

15.12　$resource 服务

我们可以将$resource服务当作自定义服务的基础。创建自定义服务给了我们对应用进行高度自定义的能力，可以对远程服务通信进行抽象，并且从控制器和视图中解耦出来。

最后，我们强烈建议在自定义的服务对象内部使用$resource。这不仅可以将加载远程服务抽象成一个独立的AngularJS服务，同时将其从控制器中解耦，保证控制器的代码清洁。另外，还使得我们可以不必关心控制器是如何取得数据的。

AngularJS对象内部的这种解耦方式同样对测试有益，因为我们可以将后端请求的结果进行储存和模拟，而不用担心在测试时真的会将请求发送给后端。

要创建一个封装$resource的服务，需要将$resource的服务注入到我们用来封装的服务对象中，并像平时一样调用其中的方法。

如下所示：

```
angular.module('myApp', ['ngResource'])
.factory('UserService', [
'$resource', function($resource) {

    return $resource('/api/users/:id', {
        id: '@'
    }, {
        update: {
            method: 'PUT'
        }
    });
}]);
```

$resourceAPI

通过$resource()方法来使用$resource服务。这个方法可以接受三个参数。

- url（字符串）

我们在这里传入一个包含所有参数的，用来定位资源的参数化URL字符串模板（参数以:符号为前缀）。对URL中的每个参数，都可以通过它们的名字来为其赋值：

```
$resource('/api/users/:id.:format', {
    format: 'json',
    id: '123'
});
```

这里需要注意，如果:之前的参数是空的（上面例子中的:id），那么URL中的这部分会被压缩成一个.符号。

15

> 如果我们使用的服务器要求在URL中输入端口号，例如`http://localhost:3000`，我们必须对URL进行转义。这种情况下URL规则看起来是这样的：`$resource('http://localhost\\:3000/api/users/:id.json')`。

- `paramDefaults`（可选，对象）

第二个参数中包含了发送请求时URL中参数的默认值。对象中的键会与参数名进行匹配。如果我们传入了一个没有在URL中设置过的参数，那它会以普通的查询字符串的形式被发送。

例如，如果URL字符串具有`/api/users/:id`这样的签名，并且我们将默认值设置为`{id: '123', name: 'Ari' }`，那么URL最终会被转换成`/api/users/123?name=Ari`。

这里可以像上面一样硬编码一个默认值来传入一个静态值，也可以设置它从一个数据对象中读取动态值。

如果要设置动态值，需要在值之前加上`@`字符作为前缀。

- `actions`（可选，对象）

动作对象是具有自定义动作，并且可以对默认的资源动作进行扩展的hash对象。

在这个对象中，对象的键就是自定义动作的名字，而`$http`设置对象的值会对URL中相应的参数进行替换。

例如，我们可以用如下形式在资源上定义一个新的`update`动作：

```
$resource('/api/users/:id.:format', {
    format: 'json',
    id: '123'
}, {
    update: {
        method: 'PUT'
    }
});
```

15.13　使用 Restangular

尽管AngularJS本身非常强大，可通过将所有的重要数据打包在应用内部而形成一个独立的应用，但这样就会错过这个框架最优秀的功能之一：和外部世界通信的能力。

本节会深入讨论一个非常不可思议的、设计良好的库：Restangular。

15.14　Restangular 简介

Restangular是一个专门用来从外部读取数据的AngularJS服务。

为什么不用`$http`或`$resource`？尽管`$http`和`$resource`是AngularJS的内置服务，但这两个服务在某些方面的功能是有限的。Restangular通过完全不同的途径实现了XHR通信，并提供了良好的使用体验。

使用Restangular能带来的所有好处在Restangular的README[1]文件中都有详细说明，这里我们简单介绍几个。

1. promise

Restangular支持promise模式的异步调用，使用起来更符合AngularJS的习惯。可以像使用原始的`$http`方法一样对响应进行链式操作。

2. promise展开

也可以像使用`$resource`服务一样使用Restangular，通过很简单的方式同时操作promise和对象。

3. 清晰明了

Restangular库几乎没有复杂或神奇的东西，无需通过猜测或研究文档就可以知道它是如何工作的。

4. 全HTTP方法支持

Restangular支持所有的HTTP方法。

5. 忘记URL

`$resource`要求明确的指定想要拉取数据的URL，Restangular并不需要事先知道URL或提前指定它们（除基础URL外）。

6. 资源嵌套

Restangular可以直接处理嵌套的资源，无需创建新的Restangular实例。

7. 一个实例

同`$resource`不同，使用过程中仅需要创建一个Restangular资源对象的实例。

15.15　安装 Restangular

要方便地安装Restangular，有多个不同的选择。可以手动从GitHub[2]下载文件，保存到本地后在页面中引用。如果本地保存路径为js/vendor目录，HTML中应该像下面这样引入这个文件：

```
<script type="test/javascript" src="js/vendor/restangular.min.js"></script>
```

也可以在页面中引用托管在jsDelivr[3]上的JavaScript库文件：

```
<script type="text/javascript"
    src="http://cdn.jsdelivr.net/restangular/latest/restangular.js"></script>
<script type="text/javascript"
    src="http://cdn.jsdelivr.net/restangular/latest/restangular.min.js"></script>
```

如果项目中使用了npm，也可以通过npm来安装Restangular：

```
$ npm install restangular
```

① https://github.com/mgonto/restangular
② https://raw.github.com/jimaek/jsdelivr/master/files/restangular/latest/restangular.min.js
③ http://www.jsdelivr.com/

当然如果项目中使用了Bower，也可以用Bower来安装：

```
$ bower install restangular
```

Restangular依赖Lo-Dash或Underscore，因此为了确保Restangular可以正常运行，需要引入这两个库中的一个。

如果通过Bower安装Restangular，Lo-Dash会同时被自动下载。

否则，可以通过jsDelivr[①]来引用`lodash`:

```
<script type="text/javascript"
    src="//cdn.jsdelivr.net/lodash/2.1.0/lodash.compat.min.js">
</script>
```

也可以从http://lodash.com/上下载Lo-Dash或通过Bower安装它:

```
bower install--savelodash
```

然后在页面里引用脚本：

```
<script type="text/javascript"
    src="/js/vendor/lodash/dist/lodash.min.js"></script>
```

同其他的AngularJS库一样，我们需要将`restangular`资源当作依赖加载进应用模块对象。

```
angular.module('myApp', ['restangular']);
```

完成后，就可以将`Restangular`服务注入到AngularJS对象中：

```
angular.module('myApp', [])
.factory('UserService', ['Restangular', function(Restangular) {
    // 现在我们已经在UserService中访问了Restangular
}]);
```

15.16　Restangular 对象简介

通过Restangular有两种方式创建拉取数据的对象。可以为拉取数据的对象设置基础路由：

```
var User = Restangular.all('users');
```

这样设置Restangular服务会让所有的HTTP请求将/users路径作为根路径来拉取数据。例如，调用上述对象的`getList()`方法会从/users拉取数据：

```
var allUsers = User.getList(); // GET /users
```

当然也可以通过单个对象来发送嵌套的请求，可以用唯一的ID来代替路由发送请求：

```
var oneUser = Restangular.one('users', 'abc123');
```

上面代码的效果是调用`oneUser`上的`get()`时向`/users/abc123`发送请求。

```
oneUser.get().then(function(user) {
```

① http://www.jsdelivr.com/#!lodash

```
    // GET /users/abc123/inboxes
    user.getList('inboxes');
});
```

从上面可以看出，Restangular非常聪明，知道如何根据在Restangular源对象上调用的方法来构造URL。但设置拉取数据的URL是很方便的，特别是当后端不支持纯粹的RESTful API时。

通过向allUrl方法传入一个独立的参数来指定请求的URL：

```
// 搜索的所有URL都将使用
// `http://google.com/`asthebaseUrl
var searches =
    Restangular.allUrl('one', 'http://google.com/');
// 将发送一个请求到GET http://google.com/
searches.getList();
```

另外也可以通过oneURL方法针对特定的请求，设置基础URL而不是操作整个请求：

```
var singleSearch =
    Restangular.oneUrl('betaSearch', 'http://beta.google.com/1');

// 触发一个请求到GET http://google.com/1
singleSearch.get();
```

15.17　使用 Restangular

现在我们已经可以操作Restangular对象了，下面我们来用它发送请求吧。

当Restangular将初始化的对象返回给我们后，可以通过几种不同的方法与后端API进行交互。

假设我们创建了一个Restangular对象代表公共讨论列表：

```
var messages = Restangular.all('messages');
```

通过这个对象，可以使用getList()来获取所有信息。getList()方法返回了一个集合，其中包含了可以用来操作特定集合的方法。

```
// 所有消息都是一个将被resolve成所有消息列表的promise
var allMessages = messages.getList();
```

同样可以使用Restangular对象来创建信息。使用post()方法来创建message对象。

post方法可以接受一个必要参数，参数类型是对象，并向指定的URL发送一个POST请求。我们也可以向请求中添加查询参数和头。

```
// POST到/messages
var newMessage = {
    body: 'Hello world'
};
messages.post(newMessage);
// 或者我们将在一个元素上调用这个函数
// 以创建嵌套的资源
var message = Restangular.one('messages', 'abc123'); message.post('replies', newMessage);
```

由于Restangular返回promise对象，我们可以调用promise对象上的方法，因此我们可以在promise对象完成时运行函数。

15

Restangular返回的是增强过的promise对象，因此除了可以调用then方法，还可以调用一些特殊的方法，比如$object。$object会立即返回一个空数组（或对象），在服务器返回信息后，数组会被用新的数据填充。这对更新一个集合后，在作用域中立即重新拉取集合的场景很有用：

```
// 然后在promise中调用
messages.post(newMessage).then(function(newMsg){
    // 首先将消息设置成空数组
    // 然后一旦getList是完整的就填充它
    $scope.messages = messages.getList().$object;
}, function(errorReason)
    // 出现了一个错误
});
```

我们也可以从集合中移除一个对象。使用remove()方法可以发送一个DELETE HTTP请求给后端。通过调用集合中一个对象（或元素）的remove()方法来发送删除请求。

```
var message = messages.get(123);
message.remove(); // 发送DELETE HTTP请求
```

更新和储存对象是常见的操作。通常情况下，这种操作由HTTP PUT方法完成。Restangular通过put()方法来支持这个功能。

要更新一个对象，首先要查询这个对象，然后在实例中设置新的属性值，再调用对象的put()方法将更新保存到后端。

> 注意，在修改一个对象之前对其进行复制，然后对复制的对象进行修改和保存是一个好做法。Restangular有自己的复制版本，因此无需对一系列方法重新进行绑定。在更新对象时使用Restangular.copy()是一个比较好的实践。

现在我们已经了解了如何操作集合中的实例，下面详细介绍嵌套资源。嵌套资源是指包含在其他组件内部的组件。例如，一个特定作者所写过的所有书籍。

Restangular默认支持嵌套资源。事实上，我们可以从集合中查询出特定的嵌套资源实例。

```
var author = Restangular.one('authors', 'abc123');
// 构建一个GET到/authors/abc123/books的请求
var books = author.getList('books');
```

Restangular中另外一个酷炫的功能是不仅可以在one和all方法创造的对象上调用post、put、getList等方法，也可以在服务器返回的对象上调用。例如，我们可以在代码中首先拉取一个作者并进行展示，然后获取他的书籍列表：

```
Restangular.one('authors', 'abc123').then(function(author) {
    $scope.author = author;
});

// 然后在代码中将
// 构建一个GET到/authors/abc123/authors的请求
// 使用$scope.author，它是从服务器返回的真实对象
$scope.author.getList('books');
```

15.17.1　我的HTTP方法们怎么办

Restangular支持所有的HTTP方法。它支持GET、PUT、POST、DELETE、HEAD、TRACE、

OPTIONS和PATCH。

```
author.get(); // GET/authors/abc123
author.getList('books');// GET/authors/abc123/books
author.put(); // PUT/authors/abc123
author.post(); // POST/authors/abc123
author.remove(); // DELETE/authors/abc123
author.head(); // HEAD/authors/abc123
author.trace(); // TRACE/authors/abc123
author.options(); // OPTIONS/authors/abc123
author.patch(); // PATCH/author/abc123
```

如果后端服务器映射资源的方式和我们预期的不符，Restangular也支持自定义HTTP方法。

例如，如果我们想得到作者的传记（不是RESTful资源），可以使用customMETHOD()函数设置URL（METHOD可以被下面的方法替代：GET、GETLIST、DELETE、POST、PUT、HEAD、OPTIONS、PATCH、TRACE）：

```
// 映射一个GET到/users/abc123/biography的请求
author.customGET("biography");
// 或者带有一个新bio对象的POST
// as {body: "Ari's bio"}
// 中间的两空字段是
// 参数字段或任意自定义头部
author.customPOST({body: 'Ari\'s Bio'},// post body
    "biography", // 路由
    {},          // 自定义参数
    {});         // 自定义头部
```

15.17.2 自定义查询参数和头

每一个HTTP方法都可以自定义查询参数和头。

为了添加自定义查询参数，需要添加一个JavaScript对象，将其作为第二个参数添加到我们的方法调用中，还可以再添加一个JavaScript对象作为第三个参数。最重要的是，在资源上调用这些方法会将这两个参数作为可选参数。

使用了自定义查询参数，一个post方法看起来像这样：

```
var queryParamObj = { role: 'admin' },
    headerObj = { 'x-user': 'admin' };

messages.getList('accounts', queryParamObj, headerObj);
```

Restangular使用起来难以置信地方便，因此我们可以集中精力在构建应用上，而不需要和这些API较劲。

15

15.18 设置 Restangular

Restangular具有高度的可定制性，可以根据应用的需要进行相应的设置。每个属性都有默认值，所以我们也无需在不必要的情况下对其进行设置。

Restangular可以在几个不同的地方进行设置，比如全局或自定义服务中。

将RestangularProvider注入到config()函数中，或者将Restangular注入到一个run()函数中，用这些方式对Restangular进行设置，无论在哪里使用Restangular都可以利用这些设置。

一个决定在何处设置Restangular实例的好方法：如果设置Restangular时需要用到其他服务，那么就在run()方法中设置，否则就在config()中进行设置。

1. 设置 baseUrl

通过setBaseUrl()方法给所有后端 API 请求设置 baseUrl。例如，如果 API 的地址是 /api/vi 而不是服务器的根路径，可以进行如下设置：

```
angular.module('myApp', ['restangular'])
    .config(function(RestangularProvider) {
        RestangularProvider.setBaseUrl('/api/v1');
    });
```

2. 添加元素转换

Restangular加载了资源之后，我们可以添加资源转换器。

使用elementTransformers可以在Restangular对象被加载后为其添加自定义方法。

这个方法会在资源被加载后当作回调函数调用，在AngularJS对象中使用这些资源前可以对资源对象进行更新或修改。

```
angular.module('myApp', ['restangular'])
    .config(function(RestangularProvider) {
        // 3个参数：
        // route
        // 如果它是一个集合——布尔值（true/false）或者
        // 如果你需要这两个选项以及变换器
        // 则不发送
        RestangularProvider.addElementTransformer('authors', false, function(element) {
            element.fetchedAt = new Date();
            return element;
        });
    });
```

对于扩展数据模型或集合有跨界方法可以使用。例如，如果我们只想更新authors资源，可以用如下方法：

```
angular.module('myApp', ['restangular'])
    .config(function(RestangularProvider) {
        // 3个参数：
        // route
        // 如果它是一个集合——布尔值（true/false）或者
        // 如果你需要这两个选项以及变换器
        // 则不发送
        RestangularProvider.extendModel('authors', function(element) {
            element.getFullName = function() {
                return element.name + ' ' + element.lastName;
            };
            return element;
        });
    });
```

3. 设置responseInterceptors

Restangular可以设置响应拦截器。responseInterceptors在需要对服务器返回的响应进行转换时非常有用。例如，如果服务器返回的数据将我们需要的数据藏在了嵌套资源中，可以用responseInterceptors把这些数据挖出来。

> getList方法始终返回数组是非常重要的，如果响应中包含带有元信息和嵌套数组的对象，我们应该用responseInterceptors把它解析出来。

responseInterceptors在每个响应从服务器返回时被调用。调用时会传入以下参数。

- data：从服务器取回的数据。
- operation：使用的HTTP方法。
- what：所请求的数据模型。
- url：请求的相对URL。
- response：完整的服务器响应，包括响应头。
- deferred：请求的promise对象。

例如，下面的设置会使getList()返回一个带有元信息的数组，在这种情况下，数组中的元素就是同路由具有相同名称的属性的值。例如，向/customers发送GET请求会返回一个像{customers: []}这样的数组。

```
angular.module('myApp', ['restangular'])
    .config(function(RestangularProvider) {
        RestangularProvider.setResponseInterceptor(function(data, operation, what) {
            if (operation == 'getList') {
                var list = data[what];
                list.metadata = data.metadata;
                return list;
            }
            return data;
        });
    });
```

4. 使用requestInterceptors

Restangular同样还支持另外一种对应的操作：我们可以在将数据实际发送给服务器之前对其进行操作。

如果要在将对象发送给服务器之前对其进行操作，那么requestInterceptors非常有用。例如，我们可以直接用_id字段同MongoDB进行通信，所以在使用PUT操作将其发送回服务器之前需要把这个字段移除。

小提示：我们可以同时使用requestInterceptors和responseInterceptors来实现全页面范围内的加载提示。在每个请求之前开始加载提示，在收到请求后停止加载提示。

使用setRequestInterceptor()来设置requestInterceptor。这个方法可以接受下面四个参数。

- element： 发送给服务器的资源；
- operation：所使用的HTTP方法；

15

- what：所请求的数据模型；
- url：所请求的相对URL。

```
angular.module('myApp', ['restangular'])
    .config(function(RestangularProvider) {
        RestangularProvider.setRequestInterceptor(function(elem, operation, what) {
            if (operation === 'put') {
                elem._id = undefined;
                return elem;
            }
            return elem;
        });
    });
```

5. 自定义字段

Restangular支持自定义字段，这对与非服务器通信非常有用，例如，同MongoDB数据库进行通信，在这种场景中id字段不会映射到真的id上，在MongoDB中id字段实际上会映射到_id.$oid上。

```
angular.module('myApp', ['restangular'])
    .config(function(RestangularProvider) {
        RestangularProvider.setRestangularFields({
            id: '_id.$oid'
        });
    });
```

6. 通过errorInterceptors来捕获错误

通过设置错误拦截器可以捕获Restangular内的错误。通过errorInterceptor可以将错误信息在应用中进行传递。

如果errorInterceptor返回false，promise链就会被中断，并且我们的应用永远都不需要处理错误。

例如，此时是处理验证失败的好时机。任何请求如果返回了401，可以通过errorInterceptor将其捕获并将用户重定向到登录页面。

```
angular.module('myApp', ['restangular'])
    .config(function(RestangularProvider) {
        RestangularProvider.setErrorInterceptor(function(resp) {
            displayError();
            return false; // 停止promise链
        });
    });
```

7. 孤立资源设置

如果我们想加载一个没有嵌套在其他资源中的资源，可以使用setParentless设置告诉Restangular不要构造嵌套结构的URL。

```
angular.module('myApp', ['restangular'])
    .config(function(RestangularProvider) {
        RestangularProvider.setParentless(['cars']);
    });
```

setParentless()设置函数可以接受两种不同类型的参数：

- 布尔型

如果参数值为true，所有的资源都会被当作孤立资源处理，没有任何URL会进行嵌套。

- 数组

只有定义在这个数组中的资源会当作孤立资源处理，数组的元素是字符串，字符串的值是资源的标识。

8. 使用超媒体

在实践中，只通过一个切入点（主URL）来同后端服务器进行通信是非常好的做法，其他数据模型通过链接来指向相关联的资源。

Restangular通过selfLink、oneUrl和allUrl来支持这个有用的做法。

首先要设置selfLink字段。同设置ID非常类似，selfLink将路径设置为数据模型的一个属性，而数据模型通过链接同对应的资源相关联。这样我们可以知道应该将PUT或GET请求发送到哪个URL。

```
angular.module('myApp', ['restangular'])
    .config(function(RestangularProvider) {
        RestangularProvider.setRestangularFields({
            selfLink: 'link.href'
        });
    });
```

设置好后，就可以开始使用这个非常有用的功能了。

首先读取所有作者的列表，这也是应用的主路由。

```
$scope.authors = Restangular.all('authors').getList().$object;
```

基于前面的设置，每一个作者都对应一个指向自己的链接，同样还有一个指向该作者对应的书籍的URL。可以像下面这样使用这些属性：

```
var firstAuthor = authors[0];
firstAuthor.name="John";

// PUT到/authors/1988-author-1
// url在firstAuthor.link.href中
firstAuthor.put();

// GET到/books/for-author/1988-author-1
var books = Restangular.allUrl('books', firstAuthor.books.href)
    .getList().$object;
```


9. 自定义Restangular服务

最后，强烈建议将Restangular封装在一个自定义服务对象内。这样做非常有用，因为在每个自定义服务中都可以对Restangular进行独立的设置。通过使用服务可以将同服务器通信的逻辑与AngularJS对象解耦，并让服务直接处理通信的业务。

AngularJS对象内部的解耦同样对测试非常有帮助，因为我们在测试时模拟后端请求的返回数据，而无需担心会真的向后端发送请求。

通过将Restangular服务注入到工厂函数中，就可以方便地对Restangular进行封装。在工厂函数内部，使用withConfig()函数来创建自定义设置。

例如：

```
angular.module('myApp', ['restangular'])
.factory('MessageService', ['Restangular', function(Restangular) {
    var restAngular = Restangular.withConfig(function(Configurer) {
        Configurer.setBaseUrl('/api/v2/messages');
    });

    var _messageService = restAngular.all('messages');

    return {
        getMessages: function() {
            return _messageService.getList();
        }
    };
}]);
```

第 16 章 XHR实践

16.1 跨域和同源策略

浏览器在全局层面禁止了页面加载或执行与自身来源不同的域的任何脚本。

同源策略允许页面从同一个站点加载和执行特定的脚本。浏览器通过对比每一个资源的协议、主机名和端口号来判断资源是否与页面同源。站外其他来源的脚本同页面的交互则被严格限制。

跨域资源共享（Cross Origin Resource Sharing，CORS）是一个解决跨域问题的好方法，从而可以使用XHR从不同的源加载数据和资源。

幸好，除CORS以外还有几个方法可以用来从外部的数据源将数据加载到应用中。我们将详细介绍其中的两种，第三种只会简要介绍（因为它需要服务器端的额外支持）：

- JSONP
- CORS
- 服务器代理

16.2 JSONP

JSONP是一种可以绕过浏览器的安全限制，从不同的域请求数据的方法。使用JSONP需要服务器端提供必要的支持。

JSONP的原理是通过`<script>`标签发起一个GET请求来取代XHR请求。JSONP生成一个`<script>`标签并插到DOM中，然后浏览器会接管并向src属性所指向的地址发送请求。

当服务器返回请求时，响应结果会被包装成一个JavaScript函数，并由该请求所对应的回调函数调用。

AngularJS在$http服务中提供了一个JSONP辅助函数。通过$http服务的jsonp方法可以发送请求，如下所示：

```
$http
.jsonp("https://api.github.com?callback=JSON_CALLBACK") .success(function(data) {
    // 数据
});
```

当请求被发送时，AngularJS会在DOM中生成一个如下所示的`<script>`标签：

```
<script src="https://api.github.com?callback=angular.callbacks._0"
    type="text/javascript"></script>
```

注意，JSON_CALLBACK被替换成了一个特地为此请求生成的自定义函数。

当支持JSOPN的服务器返回数据时，数据会被包装在由AngularJS生成的具名函数angular.callbacks._0中。

在这个例子中，GitHub服务器会返回包含在回调函数中的JSON数据，响应看起来如下所示：

```
// 简写
angular.callbacks._0({
    'meta': {
        'X-RateLimit-Limit': '60',
        'status': 200
    },
    'data': {
        'current_user_url': 'https://api.github.com/user'
    }
})
```

当AngularJS调用指定的回调函数时会对$http的promise对象进行resolve。

> 当我们自己开发支持JSONP的后端服务时，要确保响应的数据被包含在请求所指定的回调函数中。

使用JSONP需要意识到潜在的安全风险。首先，服务器会完全开放，允许后端服务调用应用中的任何JavaScript。

不受我们控制的外部站点（或者蓄意攻击者）可以随时更改脚本，使我们的整个站点变得脆弱。服务器或中间人有可能会将额外的JavaScript逻辑返回给页面，从而将用户的隐私数据暴露出来。

由于请求是由<script>标签发送的，所以只能通过JSONP发送GET请求。并且脚本的异常也很难处理。使用JSONP一定要谨慎，同时只跟信任并可以控制的服务器进行通信。

16.3　使用 CORS

近年来，W3C制定了跨域资源共享来通过标准的方式取代JSONP。

CORS规范简单地扩展了标准的XHR对象，以允许JavaScript发送跨域的XHR请求。它会通过预检查（preflight）来确认是否有权限向目标服务器发送请求。

预检查可以让服务器接受或拒绝来自全部服务器、特定服务器或一组服务器的请求。这意味着客户端和服务端应用需要协同工作，才能向客户端或服务器发送数据。

W3C制定CORS规范时对很多细节进行了抽象，并使其对客户端开发者透明，让开发者可以像发送同域请求一样方便地发送跨域请求。

16.3.1　设置

为了在AngularJS中使用CORS，首先需要告诉AngularJS我们正在使用CORS。使用config()方法在应用模块上设置两个参数以达到此目的。

首先，告诉AngularJS使用XDomain，并从所有的请求中把X-Request-With头移除掉。

X-Request-With头默认就是移除掉的，但是再次确认它已经被移除没有坏处。

```
angular.module('myApp', [])
.config(function($httpProvider) {
    $httpProvider.defaults.useXDomain = true;
    delete $httpProvider.defaults.headers
        .common['X-Requested-With'];
});
```

现在可以发送CORS请求了。

16.3.2 服务器端CORS支持

尽管这一章不深究服务端CORS的设置（第18章才会深入讨论），但是确保服务器支持CORS是很重要的。

支持CORS的服务器必须在响应中加入几个访问控制相关的头。

- Access-Control-Allow-Origin

这个头的值可以是与请求头的值相呼应的值，也可以是*，从而允许接收从任何来源发来的请求。

- Access-Control-Allow-Credentials（可选）

默认情况下，CORS请求不会发送cookie。如果服务器返回了这个头，那么就可以通过将withCredentials设置为true来将cookie同请求一同发送出去。

如果将$http发送的请求中的withCredentials设置为true，但服务器没有返回Access-Control-Allow-Credentials，请求就会失败，反之亦然。

后端服务器必须能处理OPTIONS方法的HTTP请求。

CORS请求分为简单和非简单两种类型。

16.3.3 简单请求

如果请求使用下面一种HTTP方法就是简单请求：

- HEAD；
- GET；
- POST。

如果请求除了下面列表中的一个或多个HTTP头以外，没有使用其他头：

- Accept；
- Accept-Language ；
- Content-Language；
- Last-Event-ID；

- Content-Type；
    - application/x-www-form-urlencoded；
    - multipart/form-data；
    - text/plain。

我们把这类请求归类为简单请求，因为浏览器可以不需要使用CORS就发送这类请求。简单请求不要求浏览器和服务器之间有任何的特殊通信。

一个使用$http服务的简单CORS请求和其他简单请求看起来是下面这样的：

```
$http
.get("https://api.github.com")
.success(function(data) {
    // 数据
});
```

16.3.4 非简单请求

不符合简单请求标准的请求被称为非简单请求。如果想要支持PUT或DELETE方法，又或者想给请求设置特殊的内容类型，就需要发送非简单请求。

尽管这些请求在客户端开发者看来没什么不同，但浏览器会以不同的方式处理它们。

浏览器实际上会发送两个请求：预请求和请求。浏览器首先会向服务器发送预请求来获得发送请求的许可，只有许可通过了，浏览器才会发送真正的请求。

浏览器处理CORS的过程是透明的。

同简单请求一样，浏览器会给预请求和请求都加上Origin头。

预请求

浏览器发送的预请求是OPTIONS类型的，预请求中包含以下头信息：

- Access-Control-Request-Method

这个头是请求所使用的HTTP方法，会始终包含在请求中。

- Access-Control-Request-Headers（可选）

这个头的值是一个以逗号分隔的非简单头列表，列表中的每个头都会包含在这个请求中。

服务器必须接受这个请求，然后检查HTTP方法和头的合法性。如果通过了检查，服务器会在响应中添加下面这个头：

- Access-Control-Allow-Origin

这个头的值必须和请求的来源相同，或者是*符号，以允许接受来自任何来源的请求。

- Access-Control-Allow-Methods

这是一个可以接受的HTTP方法列表，对在客户端缓存响应结果很有帮助，并且未来发送的请求可以不必总是发送预请求。

❑ Access-Control-Allow-Headers

如果设置了Access-Control-Request-Headers头，服务器必须在响应中添加同一个头。

我们希望服务器在可以接受这个请求时返回200状态码。如果服务器返回了200状态码，真正的请求才会发出。

CORS并不是一个安全机制，只是现代浏览器实现的一个标准。在应用中设置安全策略依然是我们的责任。

AngularJS中的非简单请求与普通请求看起来没有什么区别：

```
$http
.delete("https://api.github.com/api/users/1")
.success(function(data) {
    // 数据
});
```

16.4　服务器端代理

实现向所有服务器发送请求的最简单方式是使用服务器端代理。这个服务器和页面处在同一个域中（或者不在同一个域中但支持CORS），做为所有远程资源的代理。

可以简单地通过使用本地服务器来代替客户端向外部资源发送请求，并将响应结果返回给客户端。

通过这种方式，老式浏览器不必使用需要发送额外请求的CORS（只有现代浏览器支持CORS）也能发送跨域请求，并且可以在浏览器中采用标准的安全策略。

为了实现服务器端代理，需要架设一个本地服务器来处理我们所有的请求，并负责向第三方发送实际的请求。

更多关于架设服务器端组件的内容请查看第18章。

16.5　使用 JSON

JSON是JavaScript Object Notation的简写，是一种看起来像JavaScript对象的数据交换格式。事实上，当JavaScript加载它时，它确实会被当做一个对象来解析。AngularJS也会将所有以JSON格式返回的JavaScript对象解析为一个与之对应的Angular对象。

例如，如果服务器返回以下JSON：

```
[
    {"msg": "This is the first msg", state: 1},
    {"msg": "This is the second msg", state: 2},
    {"msg": "This is the third msg", state: 1},
    {"msg": "This is the fourth msg", state: 3}
]
```

当AngularJS通过`$http`服务收到这个数据后，可以像普通JavaScript对象那样来引用其中的

数据：

```
$http.get('/v1/messages.json')
    .success(function(data, status) {
        $scope.first_msg = data[0].msg;
        $scope.first_state = data[0].state;
    });
```

16.6　使用 XML

尽管AngularJS能够以完全透明的方式来处理从服务器返回的JSON对象，我们同样可以使用其他的数据格式。

假如服务器返回的是XML而非JSON格式的数据，需要将其转换成JavaScript对象。

幸好，有不少出色的开源库可以使用，同样，某些浏览器也内置了解析器，可以帮助我们将XML格式转换成JavaScript对象。

在这里我们以X2JS库为例，这是一个非常好用的开源库，可以在这里下载[①]。

首选需要确保安装了X2JS库，我们通过Bower来安装：

```
$ bower install x2js
```

然后可以在页面中从googlecode.com或我们自己安装的Bower组件中引用这个库：

```
<script type="text/javascript"
    src="https://x2js.googlecode.com/hg/xml2json.js"></script>
<!-- 或者 -->
<script type="text/javascript"
    src="bower_components/xml2json/xml2json.js"></script>
```

创建一个工厂服务以开始使用这个轻量的XML解析器，这个服务的功能很简单，就是在DOM中解析XML[②]：

```
angular.factory('xmlParser', function() {
    var x2js = new X2JS();
    return {
        xml2json: x2js.xml2json,
        json2xml: x2js.json2xml_str
    };
});
```

借助这个轻量的解析服务，可以将$http请求返回的XML解析成JSON格式，如下所示：

```
angular.factory('Data', [$http, 'xmlParser', function($http, xmlParser) {
    $http.get('/api/msgs.xml', {
        transformResponse: function(data) {
            return xmlParser.xml2json(data);
        }
    });
});
```

① https://code.google.com/p/x2js/

② 解析XML和DOM没什么关系。——译者注

现在请求的结果被转换成了JSON对象，可以像服务器本来返回的就是JSON格式一样来使用这个对象。

16.7 使用 AngularJS 进行身份验证

大多数网络应用都有需要保护的资源，这些资源不能被公开访问，只能由可以被识别且信任的授权用户访问。这些资源可能是付款信息，也可能是管理功能。

无论保护的资源是什么，进行保护的手段都是类似的。

如何实现服务器端身份验证并不是本章的内容主旨。本章集中介绍服务器端需要做什么来支持前端实现此功能。

然后会介绍如何实现客户端的身份验证保护，以及这个流程中的一些边缘情况。

16.7.1 服务器端需求

首先必须保证服务器端API的安全性。由于我们处理的代码是未编译的，且可能是由不信任的源发送的，不能假设所有的用户都聪明到可以认识到这些潜在的风险。

下面介绍常被用来保护客户端应用的两种方法。

1. 服务器端视图渲染

如果站点所有的HTML页面都是由后端服务器处理的，可以使用传统的授权方式，由服务器端进行鉴权，只发送客户端需要的HTML。

2. 纯客户端身份验证

我们希望客户端和服务端的开发工作可以解耦并各自独立进行，且可以将组件独立地发布到生产环境中，互相没有影响。因此，需要通过使用服务器端API来保护客户端身份验证的安全，但并不依赖这些API来进行身份验证。

通过令牌授权来实现客户端身份验证，服务器需要做的是给客户端应用提供授权令牌。

令牌本身是一个由服务器端生成的随机字符串，由数字和字母组成，它与特定的用户会话相关联。

> uuid库是用来生成令牌的好选择。

当用户登录到我们的站点后，服务器会生成一个随机的令牌，并将用户会话同令牌之间建立关联，用户无需将ID或其他身份验证信息发送给服务器。

客户端发送的每个请求都应该包含此令牌，这样服务器才能根据令牌来对请求的发送者进行身份验证。

服务器端则无论请求是否合法，都会将对应事件的状态码返回给客户端，这样客户端才能做出响应。

例如，我们希望服务端对所有身份验证未通过的请求都返回401状态码。

下面的表格中是一些常用的状态码：

状　态　码	含　　义
200	一切正常
401	未授权的请求
403	禁止的请求
404	页面找不到
500	服务器错误

当客户端收到这些状态码时会做出相应的响应。

数据流程如下：

(1) 一个未经过身份验证的用户浏览了我们的站点；

(2) 用户试图访问一个受保护的资源，被重定向到登录页面，或者用户手动访问了登录页面；

(3) 用户输入了他的登录ID（用户名或电子邮箱）以及密码，接着AngularJS应用通过POST请求将用户的信息发送给服务端；

(4) 服务端对ID和密码进行校验，检查它们是否匹配；

(5) 如果ID和密码匹配，服务端生成一个唯一的令牌，并将其同一个状态码为200的响应一起返回。如果ID和密码不匹配，服务器返回一个状态码为401的响应。

对一个已经通过身份验证的用户（通过了上面5个步骤的用户），流程如下：

(1) 用户请求一个受保护的资源路径（比如他自己的账号页面）；

(2) 如果用户尚未登录，应用会将他重定向到登录页面。如果用户登录了，应用会使用该会话对应的令牌来发送请求；

(3) 服务器对令牌进行校验，并根据请求返回合适的数据。

16.7.2　客户端身份验证

前面一节定义了身份验证机制需要处理的一些行为：

- 重定向未经过身份验证的页面请求；
- 捕获所有响应状态码非200的XHR请求，并进行相应的处理；
- 在整个页面会话中持续监视用户的身份验证情况。

为了对未通过验证的用户访问受保护资源的行为进行重定向，需要能够对公共资源和受保护资源进行区分。

有下面几种方法可以将路由定义为公共或非公共。

1. 保护API访问的资源

如果想要对一个会发送受保护的API请求（例如，一个服务器可能返回401状态码的API请求）的路由进行保护，但又希望可以正常加载页面，可以简单地通过`$http`拦截器来实现。

想要创建一个`$http`拦截器并能够处理未通过身份验证的API请求，首先要创建一个拦截器

来处理所有的响应。

现在，我们在应用的.config()代码块内设置$http响应拦截器，并将$httpProvider注入其中：

```
angular.module('myApp', [])
.config(function($httpProvider) {
    // 在这里构造拦截器
});
```

这个拦截器会处理所有请求的响应以及响应错误。

```
angular.module('myApp', [])
.config(function($httpProvider) {
    // 在这里构造拦截器
    var interceptor = function($q, $rootScope, Auth) {
        return {
            'response': function(resp) {
                if (resp.config.url == '/api/login') {
                    // 假设API服务器返回的数据格式如下:
                    // { token: "AUTH_TOKEN" }
                    Auth.setToken(resp.data.token);
                }
                return resp;
            },
            'responseError': function(rejection) {
                // 错误处理
                switch(rejection.status) {
                case 401:
                    if (rejection.config.url!=='api/login')
                        // 如果当前不是在登录页面
                        $rootScope.$broadcast('auth:loginRequired');
                    break;
                case 403:
                    $rootScope.$broadcast('auth:forbidden');
                    break;
                case 404:
                    $rootScope.$broadcast('page:notFound');
                    break;
                case 500:
                    $rootScope.$broadcast('server:error');
                    break;
                }
                return $q.reject(rejection);
            }
        };
    };
});
```

这个授权拦截器会处理特定请求中一些可预见的服务器响应状态码。当拦截器捕获到401状态码，会通过$broadcasts从$rootScope开始向所有的子作用域广播此事件。

另外，拦截器会为任何返回200状态码的请求将令牌保存到/api/login登录路由中。

为了实现这个拦截器，需要让$httpProvider将这个拦截器添加到拦截器链中：

```
angular.module('myApp', [])
.config(function($httpProvider) {
    // 在这里构造拦截器
    var interceptor = function($q, $rootScope, Auth) {
```

16

```
        // ...
    };
    // 将拦截器和$http的request/response链整合在一起
    $httpProvider
    .interceptors.push(interceptor);
});
```

> 查看15.5节获取更多关于$http拦截器的内容。

2. 使用路由定义受保护资源

如果我们希望始终对某些路径进行保护，或者请求的API不会对路由进行保护，那就需要监视路由的变化，以确保访问受保护路由的用户是处于登录状态的。

为了监视路由变化，需要为$routeChangeStart事件设置一个事件监听器。这个事件会在路由属性开始resolve时触发，但此时路由还没有真的发生变化。

> 通过同拦截器协同工作，这种方式会更加有效。如果不通过拦截器检查状态码，用户依然有可能发送未经授权的请求。

通过监听器对事件进行监听，并检查路由，看它是否定义为可被当前用户访问。

首先要定义应用的访问规则。可以通过在应用中设置常量，然后在每个路由中通过对比这些常量来判断用户是否具有访问权限。

```
angular.module('myApp', ['ngRoute'])
.constant('ACCESS_LEVELS', {
    pub: 1,
    user: 2
});
```

通过把ACCESS_LEVELS设置为常量，可以将它注入到.confgi()和.run()代码块中，并在整个应用范围内使用。

下面，使用这些常量来为每个路由都定义访问级别：

```
angular.module('myApp', ['ngRoute'])
.config(function($routeProvider, ACCESS_LEVELS) {
    $routeProvider
        .when('/', {
            controller: 'MainController',
            templateUrl: 'views/main.html',
            access_level: ACCESS_LEVELS.pub
        })
        .when('/account', {
            controller: 'AccountController',
            templateUrl: 'views/account.html',
            access_level: ACCESS_LEVELS.user
        })
        .otherwise({
            redirectTo: '/'
        });
});
```

上面每一个路由都定义了自身的access_level，可以根据这一点判断当前用户的授权状态，以及用户的级别是否有权限访问当前路由。

此时，用户可能处于以下两种状态：

- 未经过身份验证的匿名用户；
- 通过身份验证的已知用户。

为了验证用户的身份，需要创建一个服务来对已经存在的用户进行监视。同时需要让服务能够访问浏览器的cookie，这样当用户重新登录时，只要会话有效就无需再次进行身份验证。

这个小服务包含了一些操作用户对象的辅助函数：

```
angular.module('myApp.services', [])
.factory('Auth',  function($cookieStore,ACCESS_LEVELS) {
    var _user = $cookieStore.get('user');

    var setUser = function(user) {
        if (!user.role || user.role < 0) {
            user.role = ACCESS_LEVELS.pub;
        }
        _user = user;
        $cookieStore.put('user', _user);
    };

    return {
        isAuthorized: function(lvl) {
            return _user.role >= lvl;
        },
        setUser: setUser,
        isLoggedIn: function() {
            return _user ? true : false;
        },
        getUser: function() {
            return _user;
        },
        getId: function() {
            return _user ? _user._id : null;
        },
        getToken: function() {
            return _user ? _user.token : '';
        },
        logout: function() {
            $cookieStore.remove('user');
            _user = null; }
        }
    };
});
```

现在，当用户已经通过身份验证并登录后，可以在$routeChangeStart事件中对其有效性进行检查。

16

```
angular.module('myApp', [])
.run(function($rootScope, $location, Auth) {
    // 给$routeChangeStart设置监听
    $rootScope.$on('$routeChangeStart', function(evt, next, curr) {

        if (!Auth.isAuthorized(next.$$route.access_level)) {
```

```
            if (Auth.isLoggedIn()) {
                // 用户登录了，但没有访问当前视图的权限
                $location.path('/');
            } else {
                $location.path('/login');
            }
        }
    });
});
```

3. 发送经过身份验证的请求

当我们通过了身份验证，并取回了用户的授权令牌后，就可以在向服务器发送请求时使用令牌。同前面内容介绍的一样，我们希望服务器可以根据这个唯一的令牌对用户进行验证。从服务器的角度看，当收到一个带有令牌的请求时，验证令牌的有效性是服务器的责任之一。

如果提供的令牌是合法的，且与一个合法用户是关联的状态，那服务器就会认为用户的身份是合法且安全的。

通过令牌进行身份验证的安全性取决于通信所采用的通道，因此尽可能地使用SSL连接可以提高安全性。

如果用户已经通过了身份验证，可以在发送请求时单独给每个请求都加入验证信息，或者把令牌附加到所有的请求中。

手动使用身份令牌 手动创建一个可以发送令牌的请求，只要将token当作参数或请求头添加到请求中即可。例如，如果我们想对服务器发出一个请求，此时我们正在这个服务器上通过Backend服务请求用户分析数据：

```
angular.module('myApp', [])
.service('Backend', function($http, $q, $rootScope, Auth) {
    this.getDashboardData = function() {
        $http({
            method: 'GET',
            url: 'http://myserver.com/api/dashboard'
        }).success(function(data) {
            return data.data;
        }).catch(function(reason) {
            $q.reject(reason);
        });
    };
});
```

简单地将token当作参数（或请求头）发送就可以进行令牌验证：

```
angular.module('myApp', [])
.service('Backend', function($http, $q, $rootScope, Auth) {
    this.getDashboardData = function() {
        $http({
            method: 'GET',
            url: 'http://myserver.com/api/dashboard',
            params: {
                token: Auth.getToken()
        }).success(function(data) {
            return data.data;
        }).catch(function(reason) {
            $q.reject(reason);
```

```
        });
    };
});
```

当向后端发送请求时，请求会被添加token参数。

自动添加身份令牌 更进一步，如果想要为每个请求都添加上当前用户的令牌，可以创建一个请求拦截器，并将令牌当作参数添加进请求中。

创建请求拦截器的方法和前面创建响应拦截器的方法类似，只要将拦截目标从response换成request即可。

```
angular.module('myApp', [])
.config(function($httpProvider) {
    // 在这里构造拦截器
    var interceptor = function($q, $rootScope, Auth) {
        return {
            'request': function(req) {
                return req;
            },
            'requestError': function(reqErr) {
                return reqErr;
            }
        };
    };
});
```

在请求拦截器内部可以加入向请求中添加token参数的业务逻辑，通过用户是否持有令牌来检查身份验证情况，同时需要确保不会将手动添加的同名参数覆盖。

```
function($q, $rootScope, Auth) {
    return {
        'request': function(req) {
            req.params = req.params || {};
            if (Session.isAuthenticated() && !req.params.token) {
                req.params.token = Auth.getToken();
            }
            return req;
        },
        // ...
```

16.8 和 MongoDB 通信

即使没有后端服务，我们依然可以直接同提供了RESTful接口的数据库进行通信。

可以直接同Mongo进行通信，而无需创建后端服务。

> 在这个例子中，我们使用MongoLab[①]，这是一个SAAS服务，提供了可管理的MongoDB实例。

为了同MongoDB通信，首先需要针对Restangular对象进行一些自定义配置。

① https://mongolab.com/

下面的配置会改变全局的Restangular对象。如果我们想将设置封装起来，针对单个数据库进行配置，就需要创建一个服务，将自定义的Restangular对象封装起来。

首先设置API密钥，鉴于这个密钥在整个应用中都是不变的，可以将它设置成常量。

```
angular.module('myApp', ['restangular'])
.constant('apiKey', 'YOUR_API_KEY');
```

现在这个API密钥可以被注入到应用的任何部分当中。接下来在模块的`config()`代码块中进行设置。

为了使用MongoLab，需要将`baseUrl`设置成API的切入点：

```
//...
.config(function(RestangularProvider,apiKey) {
    RestangularProvider
    .setBaseUrl('https://api.mongolab.com/api/1/databases/YOURDB/collections');
});
```

接下来，任何发送给后端数据库的请求都需要设置API密钥。通过Restangular的`setDefaultRequestParams()`方法可以方便地进行设置：

```
//...
.config(function(RestangularProvider,apiKey) {
    //...
    RestangularProvider
    .setDefaultRequestParams({
        apiKey: apiKey
    });
});
```

接下来需要更新Restangular中的字段映射，将MongoDB的`_id.$oid`字段映射到Restangular的`id`字段上。通过`setRestangularFields()`函数可以方便地实现这个需求：

```
//...
.config(function(RestangularProvider,apiKey) {
    //...
    RestangularProvider.setRestangularFields({
        id: '_id.$oid'
    });
});
```

最后需要覆盖`_id`字段，这个字段是MongoDB在更新记录时设置的。Mongo不允许覆盖`_id`字段，所以我们通过Restangular来模拟这个过程。鉴于Restangular会调用路由来更新元素，我们不需要担心对象无法被覆盖。

```
.config(function(RestangularProvider,apiKey) {
    //...
    RestangularProvider.setRequestInterceptor(function(elem, operation, what) {
        if (operation === 'put') {
            elem._id = undefined;
            return elem;
        }
        return elem;
    });
});
```

为了保证完整性，下面是完整的配置代码：

```
angular.module('myApp', ['restangular'])
.constant('apiKey', 'API_KEY')
.config(function(RestangularProvider, apiKey) {
RestangularProvider.setBaseUrl('https://api.mongolab.com/api/1/databases/YOURDB/collections');
    RestangularProvider.setDefaultRequestParams({
        apiKey: apiKey
    });
    RestangularProvider.setRestangularFields({
        id: '_id.$oid'
    });
    RestangularProvider.setRequestInterceptor(function(elem, operation, what) {
        if (operation === 'put') {
            elem._id = undefined;
            return elem;
        }
        return elem;
    });
});
```

第 17 章 promise

Angular事件系统（我们会在第24章讨论它）给Angular应用提供了很多强大功能。它给我们的最强大功能之一是promise的自动执行。

17.1 什么是 promise

promise是一种用异步方式处理值（或者非值）的方法。promise是对象，代表了一个函数最终可能的返回值或者抛出的异常。在与远程对象打交道时，promise会非常有用，可以把它们看作远程对象的一个代理。

习惯上，JavaScript使用闭包或者回调来响应非同步的有意义的数据，比如页面加载之后的XHR请求。我们可以跟数据进行交互，就好像它已经返回了一样，而不需要依赖于回调函数的触发。

回调已经被使用了很长时间，但开发人员用它时都会很痛苦。回调使得调用不一致，得不到保证，当依赖于其他回调时，它们篡改代码的流程，通常会让调试变得非常难。每一步调用之后，都需要显式处理错误。

在执行异步方法时触发一个函数，然后期待一个回调能运行起来。与之不同的是，promise提供了另外一种抽象：这些函数返回promise对象。

例如，在传统的回调代码中，我们可能会有一个方法，用户使用该方法向他的朋友发送数据。

```
// 示例回调代码
User.get(fromId, {
    success: function(err, user) {
        if (err) return {error: err};
        user.friends.find(toId, function(err, friend) {
            if (err) return {error: err};
            user.sendMessage(friend, message, callback);
        });
    },
    failure: function(err) {
        return {error: err}
    }
});
```

这个回调金字塔已经失控了，而且我们还没有加入健壮的错误处理代码。此外，在被调用的回调内部，也需要知道参数的顺序。

刚才基于promise版本的代码看上去更接近于：

```
User.get(fromId)
    .then(function(user) {
        return user.friends.find(toId);
    }, function(err) {
    // 没找到用户
})
.then(function(friend) {
    return user.sendMessage(friend, message);
}, function(err) {
    // 用户的朋友返回了异常
})
.then(function(success) {
    // user was sent the message
}, function(err) {
    // 发生错误了
});
```

代码不仅仅是可读性变高了，也更容易理解了。我们可以保证回调是一个值，而不用处理回调接口。

注意，在第一个例子中，我们需要用跟处理正常状况不同的方式去处理异常。需要确定什么时候使用回调来处理错误,在一个传统的API响应函数签名（惯例的方法签名通常是`(err, data)`）中检查错误是否已定义。我们所有的API方法都需要实现同样的结构。

在第二个例子里，我们用同样的方式处理成功和错误。合成对象将会以常见的方式接收到错误。promise API就是用于明确地执行或者拒绝promise的，所以不必担心我们的方法实现了不同的方法签名。

17.2　为什么使用 promise

使用promise的附带收获之一是逃脱了回调地狱。promise让异步函数看上去像同步的。基于同步函数，我们可以按照预期来捕获返回值和异常值。

可以在程序中的任何时刻捕捉错误，并且绕过依赖于程序异常的后续代码。我们不需要思考这个同步代码带来的好处，就已经达到上述目的了 —— 它就在代码的本质中。

因此，使用promise的目的是：获得功能组合和错误冒泡（error bubbling）能力的同时，保持代码异步运行的能力。

promise是头等对象，自带了一些约定。

- 只有一个resolve或者reject会被调用到：
  - resolve被调用时，带有一个履行值；
  - reject被调用时要带一个拒绝原因。
- 如果promise被执行或者拒绝了，依赖于它们的处理程序仍然会被调用；
- 处理程序总是会被异步调用。

此外，可以把promise串起来，并且允许代码以通常运行的方式来处理。从一个promise冒出的异常会贯穿整个promise链。

promise总是异步执行的，可以放心使用，无需担心它们会阻塞应用的其他部分。

17.3　Angular 中的 promise

Angular的事件循环给予了Angular特有的能力，能在`$rootScope.$evalAsync`阶段中执行promise（关于运行循环的更多细节，参见第24章）。promise会坐等`$digest`运行循环结束。

这件事让我们能毫无压力地把promise的结果转换到视图上。它也能让我们不加思考地把XHR调用的结果直接赋值到`$scope`对象的属性上。

我们来建一个例子，从GitHub上返回一组针对AngularJS的开放`pull`请求。

来玩玩吧[①]。

```
<h1>Open Pull Requests for Angular JS</h1>

<ul ng-controller="DashboardController">
  <li ng-repeat="pr in pullRequests">
    {{ pr.title }}
  </li>
</ul>
```

如果有个服务返回了一个promise（第19章会深入探讨），可以在`.then()`方法中与这个promise交互，它允许我们修改作用域上的任意变量，放置到视图上，并且期望Angular会为我们执行它：

```
angular.module('myApp', [])
.controller('DashboardController', [
    '$scope', 'GithubService',
        function($scope, UserService) {
            // GithubService的getPullRequests()方法
            // 返回了一个promise
            User.getPullRequests(123)
             .then(function(data) {
                $scope.pullRequests = data.data;
        });
}]);
```

当对`getPullRequests`的异步调用返回时，在`.then()`方法中就可以用`$scope.pullRequests`这个值了，然后它会更新`$scope.pullRequests`数组。

如何创建一个 promise

想要在Angular中创建promise，可以使用内置的$q服务。$q服务在它的deferred API中提供了一些方法。

首先，需要把$q服务注入到想要使用它的对象中。

```
angular.module('myApp', [])
  .factory('GithubService', ['$q', function($q) {
  // 现在就可以访问到$q库了
}]);
```

要创建一个deferred对象，可以调用`defer()`方法：

① http://jsbin.com/UfotanA/4/edit

```
var deferred = $q.defer();
```

deferred对象暴露了三个方法，以及一个可以用于处理promise的promise属性。

- resolve（value）

resolve函数用这个值来执行deferred promise。

```
deferred.resolve({name: "Ari", username: "@auser"});
```

- reject（reason）

这个方法用一个原因来拒绝deferred promise。它等同于使用一个“拒绝”来执行一个promise。

```
deferred.reject("Can't update user");
// 等同于
deferred.resolve($q.reject("Can't update user"));
```

- notify（value）

这个方法用promise的执行状态来进行响应。

例如，如果我们要从promise返回一个状态，可以使用notify()函数来传送它。

假设我们想要从一个promise创建多个长时间运行的请求。可以调用notify函数发回一个过程通知：

```
.factory('GithubService', function($q, $http) {
    // 从仓库获取事件
    var getEventsFromRepo = function() {
        // 任务
    }
    var service = {
        makeMultipleRequests: function(repos) {
            var d = $q.defer(),
                percentComplete = 0,
                output = [];
            for (var i = 0; i < repos.length; i++) {
                output.push(getEventsFromRepo(repos[i]));
                percentComplete = (i+1)/repos.length * 100;
                d.notify(percentComplete);
            }

            d.resolve(output);

            return d.promise;
        }
    }
    return service;
});
```

有了GithubService对象上的这个makeMultipleRequests()函数，每次获取和处理一个仓库时，都会收到一个过程通知。

可以在我们对promise的使用中用到这个通知，在用promise时加上第三个函数调用。例如：

```
.controller('HomeController',
    function($scope, GithubService) {
        GithubService.makeMultipleRequests([
            'auser/beehive', 'angular/angular.js'
```

```
        ])
        .then(function(result) {
            // 处理结果
        }, function(err) {
            // 发生错误了
        }, function(percentComplete) {
            $scope.progress = percentComplete;
        });
});
```

可以在deferred对象上以属性的方式访问promise：

```
deferred.promise
```

上面这个例子完整地展示了如何创建一个函数用于响应promise，看上去可能类似于下面这些GithubService上的方法。

```
angular.module('myApp', [])
    .factory('GithubService', [
        '$q', '$http',
        function($q, $http) {
            var getPullRequests = function() {
                var deferred = $q.defer();
                // 从Github获取打开的angularjs pull请求列表
                $http.get('https://api.github.com/repos/angular/angular.js/pulls')
                .success(function(data) {
                    deferred.resolve(data);
                })
                .error(function(reason) {
                    deferred.reject(reason);
                })
                return deferred.promise;
            }

            return { // 返回工厂对象
                getPullRequests: getPullRequests
            };
}]);
```

现在我们就可以用promise API来跟`getPullRequests()` promise交互。

查看完整示例：http://jsbin.com/rukefimu/3/edit。

在上面这个service的实例中，可以用两种不同方式跟promise交互。

- `then(successFn, errFn, notifyFn)`

无论promise成功还是失败了，当结果可用之后，`then`都会立刻异步调用`successFn`或者`errFn`。这个方法始终用一个参数来调用回调函数：结果，或者是拒绝的理由。

在promise被执行或者拒绝之前，`notifyFn`回调可能会被调用0到多次，以提供过程状态的提示。

`then()`方法总是返回一个新的promise，可以通过`successFn`或者`errFn`这样的返回值执行或者被拒绝。它也能通过`notifyFn`提供通知。

- `catch(errFn)`

这个方法就只是个帮助函数，能让我们用`.catch(function(reason){})`取代`err`回调：

```
$http.get('/repos/angular/angular.js/pulls')
.catch(function(reason) {
        deferred.reject(reason);
});
```

- `finally(callback)`

`finally`方法允许我们观察promise的履行或者拒绝，而无需修改结果的值。当我们需要释放一个资源，或者是运行一些清理工作，不管promise是成功还是失败时，这个方法会很有用。

我们不能直接调用这个方法，因为`finally`是IE中JavaScript的一个保留字。纠结到最后，只好这样调用它了：

```
promise['finally'](function() {});
```

Angular的`$q` deferred对象是可以串成链的，这样即使是`then`，返回的也是一个promise。这个promise一被执行，`then`返回的promise就已经是resolved或者rejected的了。

> 这些promise也就是Angular能支持`$http`拦截器的原因。

`$q`服务类似于原始的Kris Kowal的Q库：

(1) `$q`是跟Angular的`$rootScope`模型集成的，所以在Angular中，执行和拒绝都很快。

(2) `$q` promise是跟Angular模板引擎集成的，这意味着在视图中找到的任何promise都会在视图中被执行或者拒绝。

(3) `$q`很小，所以没有包含Q库的完整功能。

17.4　链式请求

`then`方法在初始promise被执行之后，返回一个新的派生promise。这种返回形式给了我们一种特有的能力，把另一个`then`接在初始的`then`方法结果之后。

```
 // 一个响应promise的服务
 GithubService.then(function(data) {
     var events = [];
     for (var i = 0; i < data.length; i++) {
        events.push(data[i].events);
     }
     return events;
 }).then(function(events) {
     $scope.events = events;
});
```

在本例中，我们可以创建一个执行链，它允许我们中断基于更多功能的应用流程，可以籍此导向不同的结果。这个中断能让我们在执行链的任意时刻暂停或者推迟promise的执行。

> 这个中断也是`$http`服务实现请求和响应拦截器的方式。

`$q`库自带了几个不同的有用方法。

17.4.1 all(promises)

如果我们有多个promise，想要把它们合并成一个，可以使用$q.all(promises)方法来把它们合并成一个promise。这个方法带有一个参数。

- promises（数组或者promise对象）

一个promise数组或者promise的hash。

all()方法返回单个promise，会执行一个数组或者一个散列的值。每个值会响应promise散列中的相同序号或者键。如果任意一个promise被拒绝了，结果的promise也会被拒绝。

17.4.2 defer()

defer()方法创建了一个deferred对象，它没有参数，返回deferred对象的一个实例。

17.4.3 reject(reason)

这个方法创建了一个promise，被以某一原因拒绝执行了。它专门用于让我们能在一个promise链中转发拒绝的promise，类似JavaScript中的throw。在同样意义上，我们能在JavaScript中捕获一个异常，也能够转发这个拒绝，我们需要把这个错误重新抛出。可以通过$q.reject(reason)来做到这点。

这个方法带有单个参数：

- reason（常量、字符串、异常、对象）

拒绝的原因。

reject()方法返回一个已经用某个原因拒绝的promise。

17.4.4 when(value)

when()函数把一个可能是值或者能接着then的promise包装成一个$q promise。这样我们就能处理一个可能是也可能不是promise的对象。

when()函数有一个参数：

- value

该参数是个值，或者是promise

when()函数返回了一个promise，我们可以像使用其他promise一样使用它。

第 18 章

服务器通信

18

Angular最强大的组件之一是与服务端通信的能力。不管用什么后端，Angular都能通过API与其通信。

本章关注两种后端：将要开发的自定义后端，以及将后端用作服务的无服务器后端。

18.1　自定义服务器端

本节重点介绍用NodeJS构建一个自定义的服务端应用程序。尽管我们关注的是用Node构建这个服务端应用，也可以用其他任意支持HTTP API路由的服务端语言来创建后端。

> 如果你是Ruby on Rails开发人员，可以参考我们写的那本专门介绍如何使用Rails的书：Riding Rails with AngularJS[1]。

要启动用Node做后端的应用，需要先安装NodeJS。

18.2　安装 NodeJS

NodeJS是一个服务器端平台，建立在Chrome JavaScript运行时上。它是一个事件驱动的非阻塞、轻量级JavaScript运行时，能让我们在服务端编写JavaScript。

要安装NodeJS，我们可以打开nodejs.org[2]，点击大按钮Install。它会检测并且下载适合我们平台的安装程序。

> 如果因为某些原因下载了错误的包，也没什么问题。我们可以点击Downloads按钮，然后手动选择合适的包。

现在可以运行安装程序了，让它自然运行。安装完成后，就能在命令行运行两个包了：

- node
- npm

node是Node的二进制文件，我们用它来运行Node应用；而npm是Node包管理器，我们用它

① http://www.fullstack.io/edu/angular/rails

② http://nodejs.org

来安装Node库。

18.3　安装 Express

我们要用一种叫做expressjs的Web应用程序框架，它会给我们一些处理HTTP的语法糖。它允许我们只使用Web应用程序的功能，而不需要处理Node的HTTP服务器的细节。

它的特性很多，包括提供了一个干净的路由语法、动态的中间件和大量专门为Express创建的开源包。此外，许多知名企业在生产中也使用它。

为了安装Express，我们会使用npm二进制：

```
$ npm install -g express-generator
```

我们使用-g标记来将包进行全局安装。如果不想全局安装它，可以省略它，这样它会被安装在当前目录的node_modules/目录下。然而我们还是推荐对它进行全局安装。

现在，可以用Express生成器来生成Express应用。

```
$ express myApp
```

这行代码生成了一个很基本的Express应用，它带有一套依赖项，以及一个非强制性的目录结构，如图18-1所示。

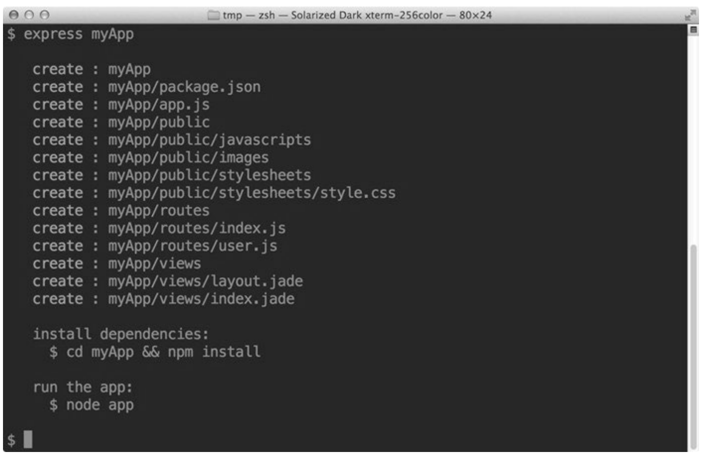


图18-1　运行Express生成器

要运行我们的应用，需要用npm把基本的依赖项安装在本地。这次，我们会用它把package.json中设置的依赖项安装到本地。

```
$ cd myApp && npm install -d
```

-d标志告诉npm把依赖项安装在本地。这个语法十分明确：可以丢开这个-d，因为它被设置为默认将依赖项安装到本地。

现在，我们来运行一下这个应用，以确认一切都能按照我们设想的那样运行。只需用node可执行文件把它运行一下就可以了，如图18-2所示。

```
$ node app.js
```



图18-2 运行Express

如果我们在Web浏览器中打开URL http://localhost:3000，刚生成的Express应用的默认页就会显示出来。

每次我们对app.js文件作修改，都需要停止服务器并且重启。在开发过程中，这个过程太麻烦了，所以我们建议不使用node.js，而是使用nodemon服务器。

要安装nodemon，需再次使用npm：

```
$ npm install --save-dev nodemon
```

--save-dev标记告诉npm把这个包保存在package.json中的devDependencies段落。我们推荐使用这种做法，因为当团队有多个开发人员时，这会有所帮助：可以确保整个团队对代码库都有正确的依赖项。

我们可以不用node app.js启动应用，而用下面的代码替换它：

```
$ nodemon app.js
```

每次对app.js文件作修改并且保存时，nodemon会自动重启Node应用。

该应用在app.js中启动。在app.js文件中，有两个重要的组件值得注意：提供静态文件的静态路径和执行的路由（以及如何执行的）。

```
// ...
app.use(express.methodOverride());
app.use(app.router);
// 第一行
app.use(express.static(path.join(__dirname, 'public')));
// ...
app.get('/', routes.index);
app.get('/users', user.list);
// ...
```

18

第一行代码调用了express.static()，告诉Node到public/目录中查找任何匹配请求路由的文件。例如，如果请求的路由是/users，它就会查找一个叫“users”的文件。

第二组代码（app.get()）匹配在public/目录中不存在静态文件的路由。

要使用Angular应用，需要在生成的app.js上作两次修改：

首先，我们交换express.static()和app.router行，如下所示：

```
// ...
app.use(express.methodOverride());
// 把该行移至下一行上方
app.use(express.static(path.join(__dirname, 'public')));
app.use(app.router);
// ...
```

尽管严格来说这个交换不是很有必要，但它有助于后续对HTML5Mode的支持，并且会告诉Express预取我们之前应用中定义的public/目录下的文件。

其次，移除指向“/”路由的一行：

```
// ...
// app.get('/', routes.index); // 删除这一行
app.get('/users', users.list);
// ...
```

现在可以像平常一样在public目录中写Angular应用了。

18.4　调用 API

本地的Node服务器现在就把应用运行起来了，所以可以调用我们自己的API，我们会在Express服务器中开发这些API。

例如，我们开发一个应用来记录用户点击特定按钮的次数，需要写两个路由：

GET /hits 该路由返回我们点击按钮的总次数。

POST /hit 该路由对按钮的一次新点击进行记录，并且给我们返回最新的总点击数。

首先，我们来构建应用的index.html页面基本视图。我们会把它放在Node应用的public/中，这样如果路由请求的是/或者/index.html，Express都会用这个文件来响应请求：

```
<!doctype html>
<html lang="en" ng-app="myApp">
  <head>
    <title>Node app</title>
    <link rel="stylesheet" href="stylesheets/style.css">
    <script src="bower_components/angular/angular.min.js"></script>
  </head>
  <body>
    <div ng-controller="HomeController">
     <h3>Button hits: {{ hits }}</h3>
     <button ng-click="registerHit()">
       HIT ME, if you dare
     </button>
   </div>
```

```
    <script src="javascripts/services.js"></script>
    <script src="javascripts/app.js"></script>
  </body>
</html>
```

在public/javacscripts/app.js文件中，我们在myApp Angular模块的顶部添加一个控制器：

```
angular.module('myApp', [
    'ngRoute',
    'myApp.services'
])
.controller('HomeController', function($scope, HitService) {
    HitService.count()
        .then(function(data) {
            $scope.hits = data;
        });

   $scope.registerHit = function() {
       HitService.registerHit()
           .then(function(data) {
               $scope.hits = data;
           });
   }
});
```

我们会建立一个Angular服务来响应对这些路由的调用，就像在上面的控制器里看到的那样：

```
angular.module('myApp.services', [])
    .factory('HitService', function($q, $http) {
        var service = {
            count: function() {
                var d = $q.defer();
                $http.get('/hits')
                    .success(function(data, status) {
                        d.resolve(data.hits);
                    }).error(function(data, status) {
                       d.reject(data);
                   });
               return d.promise;
           },
           registerHit: function() {
               var d = $q.defer();
               $http.post('/hit', {})
                   .success(function(data, status) {
                       d.resolve(data.hits);
                   }).error(function(data, status) {
                       d.reject(data);
                   });
               return d.promise;
           }
       }
       return service;
});
```

有关服务的更多信息，敬请参阅第14章。

这个服务暴露了两种调用前文定义过的路由的方法：

- count
- registerHit

在Node应用的app.js中，需要注册两个新的路由，并且创建定义路由的功能。

这两个新的Node路由与我们在上面调用的服务路由相匹配：

```
// ...
var hits = require('./routes/hits');
// ...
app.get('/hits', hits.count);
app.post('/hit', hits.registerNew);
// ...
```

唯一剩下来的组件就是创建真正的后端，记录点击数的服务端逻辑。

在NodeJS中，每个请求的模块通过exports方法来暴露方法。要暴露上面提到的两个方法count和registerNew，需要把它们附加到routes/hits.js文件中的exports对象上。

在routes/hits.js文件中，我们在内存中创建一个点击仓库来存储点击数，这样，如果重启服务器的话，点击数也会被重置。

```
 /*
 * HIT service
 */
 var hits = 0;
 exports.count = function(req, res){
     res.send(200, {
         hits: hits
     });
 }
exports.registerNew = function(req, res) {
    hits += 1;
    res.send(200, {
        hits: hits
    });
}
```

现在，如果启动我们的Node应用，把路由指向http://localhost:3000，就会看到已经给自己的Angular应用添加了预期的功能，如图18-3所示。



图18-3　首次启动

按钮被点击之后如图18-4所示。



图18-4　点击按钮之后

18.5　使用 Amazon AWS 的无服务器应用

构建一个单页应用（SPA）的最大好处之一是能够组织静态文件，而不需要建立并运行一个后端基础架构的服务。

然而，我们要构建的多数应用，需要一个包含自定义数据的后端服务器的支持。有越来越多的选择，能让开发人员只需关注构建前端代码，而暂时不管后端。

Amazon最近发布了一个选项，能让我们在浏览器中创建无服务器的Web应用：Amazon AWS JavaScript SDK①。

Amazon的基于浏览器的（服务端的是用NodeJS）SDK能让我们安心地组织应用，并且与工业级的后端服务进行交互。

通过使用S3来存放应用和文件，将DynamoDB用作NoSQL存储，以及其他的海量服务，将应用全部存放于Amazon基础架构中现在已经成为可能。我们甚至可以从客户端安全地接受支付，并且从Amazon CDN中获得所有收益。

基于这个发布，JavaScript SDK现在能让我们跟5种Amazon AWS服务进行交互。这五种服务是：

18.5.1　DynamoDB

这个快速且完全受控的NoSQL数据库服务能让我们扩展到无限大小，自带多重复制和安全访问控制。

18.5.2　简单通知服务（SNS）

这个服务是一种快速灵活、完全受控的推送服务，能让我们把消息推送到移动设备和其他服务，比如email或者甚至是Amazon自己的简单队列服务（SQS）。

① http://aws.amazon.com/

18.5.3　简单队列服务（SQS，Simple Queue Service）

这个快速、可信赖、完全受控的队列服务能让我们以一种良好管理的方式创建巨型队列。我们可以创建大型请求对象，这样可以用一个通用队列把我们的应用组件从其他组件中完全解耦。

18.5.4　简单存储服务（S3）

这个著名的、完全受控的海量数据存储能让我们存储无限数量的大对象（上限是5T），对象数量不限。我们可以使用S3从各个地方来安全地存储加密的受保护数据，甚至能使用S3来托管我们的Angular应用。

18.5.5　安全令牌服务（STS）

这个Web服务允许我们为IAM用户请求临时的受限权限认证。我们不会深入探讨STS，但是它确实为创建数据之上的受限安全操作提供了一个不错的接口。

18.6　AWSJS + Angular

本节打算演示如何把应用做好，让它们迅速在AWSJS体系中运行起来。

要做到这一点，我们要先创建一个可以让客户上传屏幕截图的缩略图，即Gunroad[①]的极简版本。我们可以通过集成美妙的Stripe[②] API来让他们出售自己的截屏。

> 对这两个服务，我们已经推荐得够多了，这个迷你演示并非要用来取代他们的服务，而只是用于展示Angular和AWS API的强大。

为了创建我们的产品，需要做到以下几点：

- 允许用户登录我们的服务，存储他们唯一的email；
- 允许用户上传与他们相关的文件；
- 允许用户点击图像，并且给这些用户一个购买这个图像的选项；
- 接受信用卡的费用，并且直接从单页Angular应用接受款项。

18.7　开始

我们从一个标准结构的index.html开始：

```
<!doctype html>
<html>
  <head>
  <script
   src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.13/angular.min.js">
  </script>
  <script
   src="http://code.angularjs.org/1.2.13/angular-route.min.js"></script>
```

① https://gumroad.com/

② http://stripe.com

```
    <link rel="stylesheet" href="styles/bootstrap.min.css">
  </head>
  <body>
    <div ng-view></div>
    <script src="scripts/app.js"></script>
    <script src="scripts/controllers.js"></script>
    <script src="scripts/services.js"></script>
    <script src="scripts/directives.js"></script>
  </body>
</html>
```

在这个标准的Angular模板中，并未加载任何出格的东西。我们加载了基础的Angular库，以及ngRoute和自定义的应用代码。

我们的应用代码也是标准的。scripts/app.js文件简单地定义了一个带有单个路由/的Angular模块：

```
 angular.module('myApp', [
     'ngRoute',
     'myApp.services',
     'myApp.directives'])
     .config(function($routeProvider) {
         $routeProvider
             .when('/', {
                 controller: 'MainController',
                 templateUrl: 'templates/main.html'
            })
            .otherwise({
                redirectTo: '/'
            });
});
```

scripts/controllers.js文件从主模块创建了控制器：

```
 angular.module('myApp')
     .controller('MainController', function($scope) {

 });
```

scripts/services.js和scripts/directives.js文件也很简单，如图18-5所示。

```
// scripts/services.js
angular.module('myApp.services', []);

// scripts/directives.js
angular.module('myApp.directives', [])
```

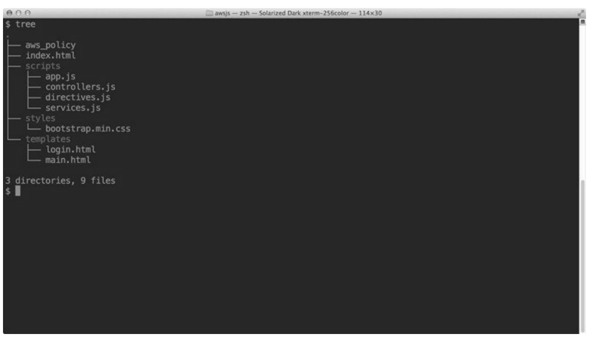


图18-5 Angular结构

18.8　介绍

AWS生态系统很庞大，在全世界各地被广泛应用于生产中。Amazon运营的大量有用服务，使它成为了一个梦幻平台，我们在这个平台上建立自己的应用。

从历史上看，这些API并不总是容易使用和理解，所以我们希望在这里解决其中一些困惑。

传统上，我们会使用一个经过认证的请求，应用则使用`client_id`或者秘密访问的key模型。既然我们是在浏览器中操作，把`client_id`和`client_secret`嵌入每个人都能看到的浏览器中是不太好的。（如果它是用明文嵌入的，就没有什么秘密可言了，对吧？）

幸好，AWS团队已经提供了一个替代方法，用于对我们的站点进行身份识别和认证，以获取对AWS资源的访问权。

创建基于AWS的Angular应用的第一步，是建立这个我们在整个过程中都会用到的相对复杂的认证和授权。

目前（写作本书时），AWS JS库与三个身份验证提供者进行了简洁整合：

- Facebook
- Google Plus
- Amazon Login

本节将关注集成Google+ API来实现我们的登录，但对其他两个身份验证提供者来说，这个过程是非常类似的。

18.9　安装

先说重要的，我们需要安装index.html中的文件。在我们的index.html中，需要包含AWS-SDK库和Google API库。

需要修改一下index.html，使它包含这些库：

```
<!doctype html>
<html>
  <head>
    <script
      src="http://code.angularjs.org/1.2.13/angular.min.js"></script>
    <script
      src="http://code.angularjs.org/1.2.13/angular-route.min.js"></script>
    <script
      src="https://sdk.amazonaws.com/js/aws-sdk-2.0.0-rc4.min.js"></script>
   <link rel="stylesheet" href="styles/bootstrap.min.css">
  </head>
  <body>
   <div ng-view></div>
   <script src="scripts/app.js"></script>
   <script src="scripts/controllers.js"></script>
   <script src="scripts/services.js"></script>
   <script src="scripts/directives.js"></script>
   <script type="text/javascript" src="https://js.stripe.com/v2/"></script>
   <script type="text/javascript">
     (function() {
```

```
        var po = document.createElement('script');
        po.type = 'text/javascript';
        po.async = true;
        po.src = 'https://apis.google.com/\js/client:plusone.js?onload=onLoadCallback';
        var s = document.getElementsByTagName('script')[0];
        s.parentNode.insertBefore(po, s);
      })();
    </script>
  </body>
</html>
```

现在，注意我们给Google JavaScript库添加了一个onload回调，并且没有使用ng-app来启动我们的应用。如果让Angular自动启动我们的应用，我们会进入一个竞态条件。在这种条件下，Google API可能在应用启动时尚未加载。

应用的这种不确定性会让体验变得无法使用，所以，我们要在onLoadCallback中手动启动应用。

为了手动启动应用，我们在window服务中添加了onLoadCallback函数。在我们能启动Angular之前，需要确认Google登录客户端已经加载。

Google API客户端，或者gapi，是在运行时间被包含进来的，并且是被默认设置的，以便延迟加载服务。通过告诉gapi.client在启动应用之前提前加载oauth2库，我们避免了任何潜在的oauth2库不可用的后果。

```
// in scripts/app.js
window.onLoadCallback = function() {
    // 当文档对象准备好了
    angular.element(document).ready(function() {
        // 启动oauth2库
        gapi.client.load('oauth2', 'v2', function() {
            // 最后，启动我们的Angular应用
            angular.bootstrap(document, ['myApp']);
        });
    });
}
```

必要的库创建好了，而且我们的应用也做好了启动的准备，这时我们可以建立应用的授权部分。

18.10　运行

鉴于我们是在使用一个服务，它取决于我们的URL是一个预期的URL，我们需要运行一个服务器，而不是简单地在浏览器中加载HTML。

我们推荐使用非常简单的Python SimpleHTTPServer来提供文件目录。这个python服务器是写来提供当前目录的，这个目录会被在本机当作一个web服务器运行。我们可以使用SimpleHTTPServer来*假装*是在web服务器上运行一个应用，并且把它加载到web浏览器中。

```
$ cd PROJECT_DIRECTORY
$ python -m SimpleHTTPServer 9000
```

现在可以在浏览器中加载URL http://localhost:9000/ ，并且看到我们的Angular应用在浏览器中运行。

18

18.11　用户认证/鉴权

首先，需要从Google获得一个`client_id`和一个`client_secret`，这样才能真正地与Google Plus登录系统进行交互。

为获取一个应用，转到Google API控制台①，并且创建一个项目，如图18-6所示。

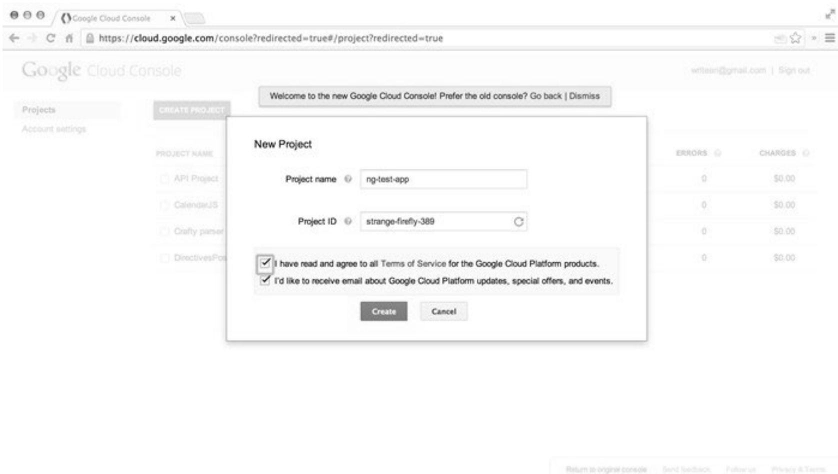


图18-6　创建一个Google Plue项目

点击名称打开项目，然后点击APIs & auth导航按钮。从这里，我们给Google+API授权。找到APIS按钮，点击它。找到Google+ API条目，并且点击OFF到ON滑块上，如图18-7所示。

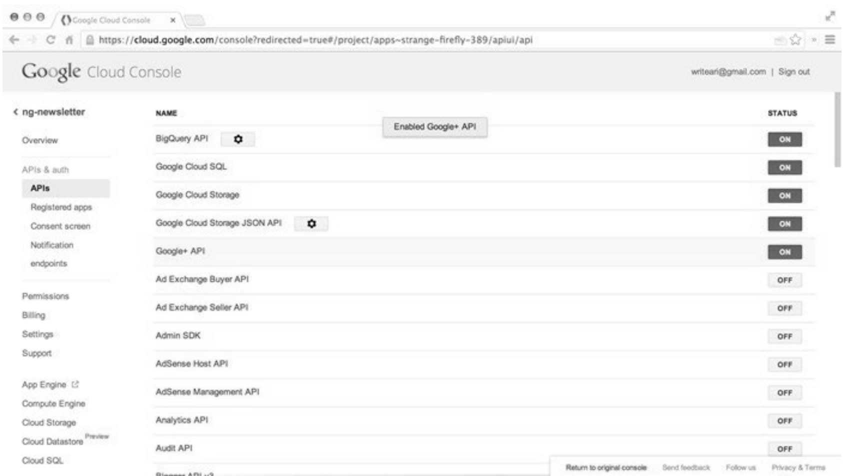


图18-7　授权于Google+ API

设置完之后，我们需要创建并且注册一个应用，并且使用它的应用ID来进行已认证的调用。

找到Registered apps选项，点击它以创建一个应用。确认在询问应用类型时选择了Web Application选项，如图18-8所示。

① https://developers.google.com/console

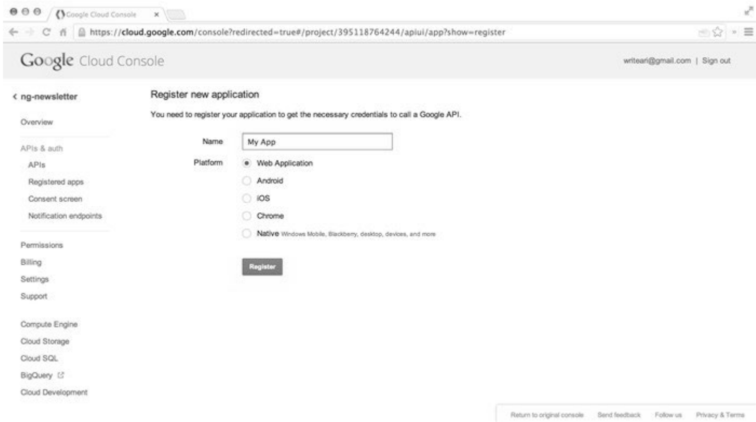


图18-8 创建一个已注册的应用

设置好了之后，我们到达了应用详情页面。选择OAuth 2.0 Client ID下拉块，记下应用的Client ID。我们很快就要使用这个ID了。

最后，把localhost源添加到应用的WEB ORIGIN中，这样保证我们能够在本地使用这个API进行开发，如图18-9所示。

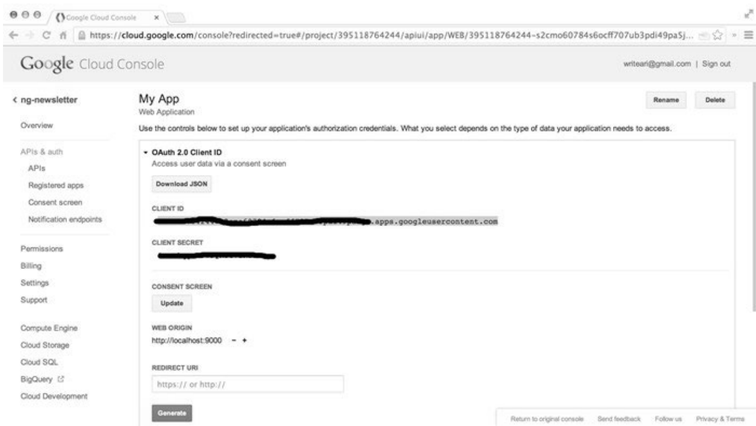


图18-9 已注册应用的详细信息

接下来，我们需要创建一个Google+登录指令。这个Angular指令能让我们用单个文件元素给应用添加一个自定义登录按钮。

> 有关指令的更多信息，请参阅第8章。

我们将要在Google登录上添加两个功能：一个会被添加到标准Google登录按钮上的元素，以及一个在按钮被渲染之后运行的自定义函数。

当要使用Google+登录指令在页面上包含一个登录元素时，我们会使用一种相当简洁的指令定义。在我们的指令中，要做下面这些事情：

(1) 创建一个DOM元素，能够设置样式（使用一个模板）；

(2) 设置在指令中需要的Google+属性；

(3) clientid；
(4) 作用域；
(5) oauth响应的回调；
(6) 设置结束登录响应的自定义回调方法；
(7) 允许指令的用户基于成功登陆定义一个自定的函数。

上面列出的我们指令的部分都是很明确的，可以在下面的完整代码中看到。上面列表中的一个组件是这个指令独有的，它允许用户定义一个方法，在成功登录之后运行。

在隔离的作用域方法中，我们会添加一个自定义函数，它能够在指令中指向所包含的作用域上定义的一个函数。为了做到这个，我们会使用&符号来告诉Angular，我们感兴趣的是绑定一个函数，而不是简单的数据。

```
scope: {
    afterSignin: '&'
}
```

有关绑定策略的更多信息，参见10.3节。

在scripts/directives.js中的最终指令如下所示：

```
angular.module('myApp.directives', [])
    .directive('googleSignin', function() {
        return {
            restrict: 'A',
            template: '<span id="signinButton"></span>',
            replace: true,
            scope: {
                afterSignin: '&'
            },
            link: function(scope, ele, attrs) {
                // 设置标准的google类
                attrs.$set('class', 'g-signin');
                // 设置clientid
                attrs.$set('data-clientid',
                    attrs.clientId+'.apps.googleusercontent.com');
                // 建立作用域的url
                var scopes = attrs.scopes || [
                    'auth/plus.login',
                    'auth/userinfo.email'
                ];
                var scopeUrls = [];
                for (var i = 0; i < scopes.length; i++) {
                    scopeUrls.push('https://www.googleapis.com/'+scopes[i]);
                };

                // 创建一个自定义回调方法
                var callbackId = "_googleSigninCallback",
                    directiveScope = scope;
                window[callbackId] = function() {
                    var oauth = arguments[0];
                    directiveScope.afterSignin({oauth: oauth});
                    window[callbackId] = null;
                };

                // 设置标准的google登录按钮的设置
                attrs.$set('data-callback', callbackId);
```

```
                attrs.$set('data-cookiepolicy', 'single_host_origin');
                attrs.$set('data-requestvisibleactions',
                    'http://schemas.google.com/AddActivity')
                attrs.$set('data-scope', scopeUrls.join(' '));

                // 最后，刷新客户端库
                // 强迫按钮在浏览器中重绘
                (function() {
                    var po = document.createElement('script');
                   po.type = 'text/javascript';
                    po.async = true;
                    po.src = 'https://apis.google.com/js/client:plusone.js';
                    var s = document.getElementsByTagName('script')[0];
                    s.parentNode.insertBefore(po, s);
                })();
            }
        }
});
```

尽管这个指令很长，但它还是相当简单的。我们是在给Google按钮赋值一个g-signin样式类，添加基于传入属性的Client ID，建立作用域，等等。

这个指令的一个特有部分是我们在窗口自定义回调函数。既然Google的库要求我们有一个回调函数，以便在登录之后作提醒用，我们需要在指令中模拟这个回调。使用window[callbackId]能让我们在调用这个函数时，模拟这个需要在JavaScript中调用的回调方法，能让我们真正调用到本地的afterSignin动作。

然后，我们将全局对象清理掉，因为在AngularJS中是比较忌讳全局状态的。

指令准备好了之后，就可以在视图中包含这个指令了。我们将在视图中调用指令，把指令上的client-id和after-signin属性替换成我们自己的，如下所示：

> 确认包含了oauth参数，正如在after-signup属性中写的那样。我们必须用这种方式调用这个参数，这是由Angular在指令中调用带参数方法的机制决定的。

```
<h2>Signin to ngroad</h2>
<div google-signin
  client-id='CLIENT_ID'
  after-signin="signedIn(oauth)"></div>
<pre>{{ user | json }}</pre>
```

> 本示例中的用户数据就是登录返回的access_token（如果登录了的话）。它不是存在我们的服务器上的，也不是敏感数据，而且会在我们离开页面时消失。

最后，我们需要让按钮实实在在地引发一个操作，所以要在控制器中定义after-signin的方法signedIn(oauth)。

这个signedIn()在真实的应用中为我们消除已认证的页面。

> 这个方法会是重定向到一个新路由的理想位置（例如，把已认证的用户导向/dashboard路由）。

```
angular.module('myApp')
    .controller('MainController',
    function($scope) {
```

18

```
        $scope.signedIn = function(oauth) {
            $scope.user = oauth;
        }
});
```

18.12　UserService

在更加深入地探讨AWS方面的内容之前，我们先要创建一个UserService，用于持有新用户。UserService会处理与AWS后端交互的多数工作，并且会保持当前用户的一个副本。

尽管还没有对添加后端这件事准备得特别充分，我们还是可以先把它构建成持有一个用户实例的持久副本。

在scripts/services.js中，我们创建UserService的开始部分：

```
angular.module('myApp.services', [])
    .factory('UserService', function($q, $http) {
        var service = {
            _user: null,
            setCurrentUser: function(u) {
                if (u && !u.error) {
                   service._user = u;
                    return service.currentUser();
                } else {
                   var d = $q.defer();
                   d.reject(u.error);
                   return d.promise;
                }
            },
            currentUser: function() {
                var d = $q.defer();
                d.resolve(service._user);
                return d.promise;
            }
        };
        return service;
});
```

尽管这个设置现在还有些人为的痕迹，但我们要的是在服务中把currentUser固化的功能。

记住，服务是单例对象，它们存在于应用的生命周期中。

至此，我们可以把用户设置给UserService，而不是简单地在signedIn()函数的返回中设置用户，如下所示：

```
angular.module('myApp')
    .controller('MainController',
        function($scope) {
            $scope.signedIn = function(oauth) {
                UserService.setCurrentUser(oauth)
                    .then(function(user) {
                        $scope.user = user;
                });
            }
});
```

要让我们的应用能运行，需要保持真实用户的email，这样我们可以提供一个更好的方法，与用户进行交互，并且通过用户保存一些持久的、特有的数据。

我们使用`gapi.client.oauth2.userinfo.get()`方法来获取用户的email地址，而不仅仅是持有用户的`access_token`（以及其他的各种访问细节）。

在`UserService`中，我们需要通过更新`currentUser()`方法来包含这个功能：

```
 // ...
 },
 currentUser: function() {
     var d = $q.defer();
     if (service._user) {
         d.resolve(service._user);
     } else {
         gapi.client.oauth2.userinfo.get()
             .execute(function(e) {
                service._user = e;
            })
    }
    return d.promise;
}
// ...
```

18.13 迁移到 AWS 上

现在，正如我们在刚开始时说的那样，需要建立带有AWS服务的鉴权。

> 如果还没有AWS账号的话，去aws.amazon.com注册一个吧，免费而且快速。

先讲重要的：创建一个IAM角色。IAM（AWS's Identity and Access Management，AWS的认证和存取管理服务）是AWS服务如此强大的原因之一。有了IAM，我们可以创建对系统和数据的细粒度访问控制。

遗憾的是，IAM的这种灵活性和强大也让它变得有些复杂，所以我们在此介入创建它的过程，并且尽量把它弄得清晰一些。

通过导航到IAM控制台[，并且点击Roles导航连接，我们来创建IAM角色。

`https://console.aws.amazon.com/iam/home?region=us-east-1#roles`

我们需要点击Create New Role按钮来给新角色一个名称，我们把它命名为google-web-role，如图18-10所示。

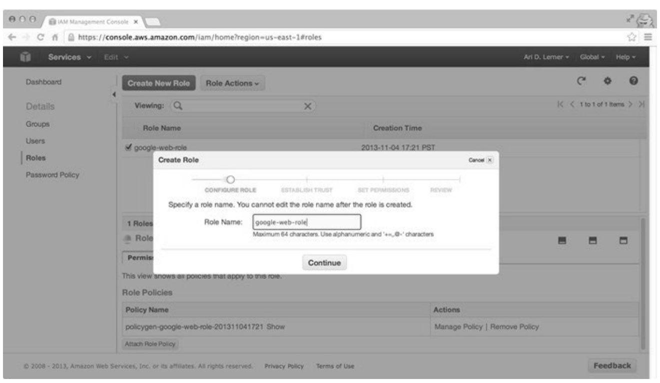


图18-10 创建一个新角色

接下来，需要把IAM角色配置为Web Identity Provider Access角色类型，这样我们可以管理新角色对AWS服务的存取，如图18-11所示。

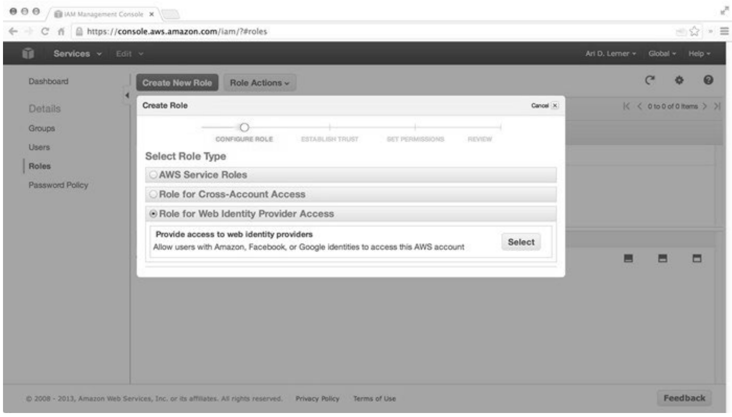


图18-11　设置角色类型

还记得我们在前面用Google创建的CLIENT ID吗？在下一个屏幕中，从下拉框中选择Google，并且把CLIENT ID粘贴到Audience框中。

这一步把我们的IAM角色和Google应用统一起来了，这样我们的应用可以用一个已认证的Google用户来访问AWS服务，如图18-12所示。

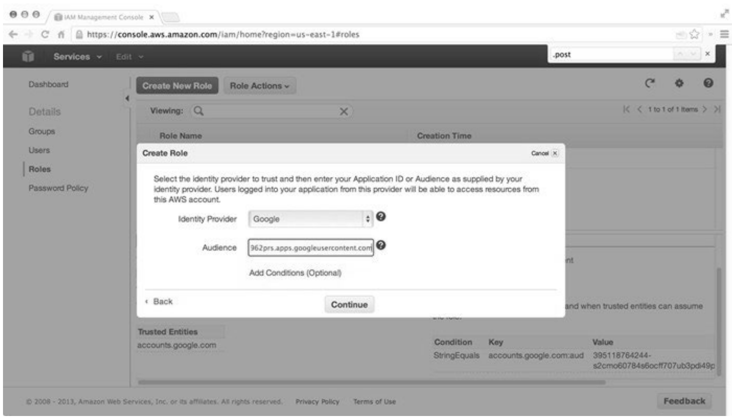


图18-12　Google认证

然后，点击Verify Trust（下一屏幕，它显示了AWS服务的原始配置），为应用创建我们的策略。

策略生成器是建立策略的快捷方式。此时，必须设置用户能进行哪些操作，不能进行哪些操作。

对于用户可能采取的行动，我们要努力说得非常具体：

1. S3

在具体的bucket上（在我们的示例中是`ng-newsletter-example`），我们将要允许用户进行以下操作：

- `GetObject`
- `ListBucket`
- `PutObject`

S3 bucket的Amazon Resource Name（ARN）如下所示：

```
arn:aws:s3:::ng-newsletter-example/*
```

2. DynamoDB

对于两个特有的表资源，我们会允许以下操作：

- `GetItem`
- `PutItem`
- `Query`

DynamoDB表的Amazon Resource Name如下所示：

```
[
    "arn:aws:dynamodb:us-east-1:<ACCOUNT_ID>:table/Users",
    "arn:aws:dynamodb:us-east-1:<ACCOUNT_ID>:table/UsersItems"
]
```

我们策略的最终版本可以在[1]找到，如图18-13所示。

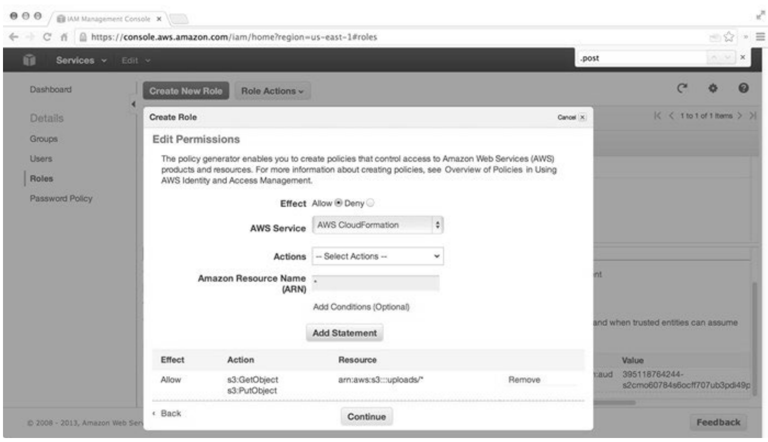


图18-13　添加IAM策略

更多有关这个令人困惑的ARN号码的信息，请查阅[2]相应的Amazon文档。

我们需要得到的最后一个信息是Role ARN。它可以在IAM控制台中的IAM用户的summary选项卡中找到。

记下这个字符串，稍后我们会设置它，如图18-14所示。

① http://d.pr/9Obg

② http://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html

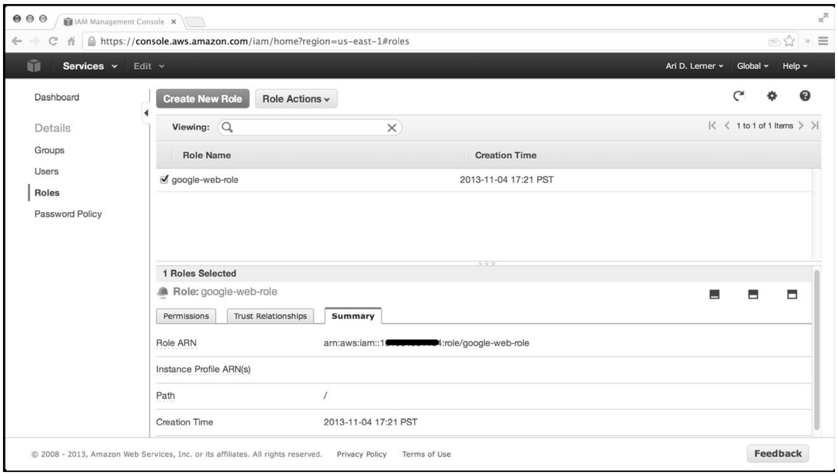


图18-14　Role ARN

鉴于我们完成了创建IAM用户的工作，可以探讨Angular应用的集成了。

18.14　AWSService

我们移动一下应用的根，把AWS集成到我们自己的服务AWSService中，然后把它构建出来。

鉴于要能够在配置阶段能对服务做一些自定义的配置，我们把它创建为一个提供者。

> 记住，唯一能注入到.config()函数的服务类型是.provider()类型。

首先，我们在scripts/services.js中创建provider的桩：

```
 // ...
 .provider('AWSService', function() {
     var self = this;
     self.arn = null;

     self.setArn = function(arn) {
         if (arn) self.arn = arn;
     }

    self.$get = function($q) {
        return {}
    }
});
```

我们已经能够看到，需要给这个服务设置Role ARN，这样能把适当的用户加到正确的服务中。

像上面一样把AWSService建立为provider，这能让我们在scripts/app.js中这样设置：

```
angular.module('myApp',
    ['ngRoute', 'myApp.services', 'myApp.directives']
)
.config(function(AWSServiceProvider) {
    AWSServiceProvider
        .setArn(
            'arn:aws:iam::<ACCOUNT_ID>:role/google-web-role');
})
```

现在，可以继续处理AWSService，不用担心会覆盖我们的Role ARN。创建这个提供者能使在不同应用之间分享变得非常容易，而不用每次都编写自定义的胶水代码。

至此，我们的AWSService还什么都没干。我们所需要做的最后一件事就是：确保是在给真正登录的用户访问权。

这个最后的步骤就是，我们要告诉AWS库，有一个已认证用户能够作为IAM角色进行操作。

我们把这个凭证创建为promise，它最终会执行，这样，我们可以定义应用的不同部分，无需为检验凭证是否已加载而烦恼，只需使用promise上的.then()方法。

我们来修改服务中的$get()方法：添加一个叫做setToken()的方法，它会创建一套新的WebIdentityCredentials：

```
// ...
self.$get = function($q) {
    var credentialsDefer = $q.defer(),
        credentialsPromise = credentialsDefer.promise;
    return {
        credentials: function() {
            return credentialsPromise;
        },
        setToken: function(token, providerId) {
           var config = {
               RoleArn: self.arn,
               WebIdentityToken: token,
               RoleSessionName: 'web-id'
           }
           if (providerId) {
               config['ProviderId'] = providerId;
           }
           self.config = config;
           AWS.config.credentials =
               new AWS.WebIdentityCredentials(config);
           credentialsDefer
               .resolve(AWS.config.credentials);
       }
    }
}
// ...
```

现在，当我们通过登录Google获取oauth.access_token时，只需把id_token传递给这个函数，它会负责AWS的配置设置。

我们用调用setToken的方法来修改UserService服务，如下所示：

```
// ...
.factory('UserService', function($q, $http) {
    var service = {
        _user: null,
        setCurrentUser: function(u) {
            if (u && !u.error) {
                AWSService.setToken(u.id_token);
                return service.currentUser();
            } else {
               var d = $q.defer();
               d.reject(u.error);
               return d.promise;
```

```
            }
    },
// ...
```

18.15　在 Dynamo 上开始

在应用中，我们想把一个用户上传的所有图像都关联到这个唯一的用户。要创建这个关联，我们会创建一个Dynamo表存储用户，另外一个存储用户和用户上传文件的关联关系。

要开始跟Dynamo打交道，首先要初始化一个Dynamo对象。我们在`AWSService`服务对象中做这件事，就像这样：

```
 // ...
 setToken: function(token, providerId) {
     // ...
 },
 dynamo: function(params) {
     var d = $q.defer();
     credentialsPromise.then(function() {
         var table = new AWS.DynamoDB(params);
         d.resolve(table);
    });
    return d.promise;
},
// ...
```

正如之前讨论的那样，通过在服务对象中使用promise，只需使用promise的`.then()` API方法来确保凭证在使用之初就已经被设置好了。

你可能会问，为什么我们要用`dynamo`方法来设置参数。有时，我们会要用不同的配置和不同的设置跟DynamoDB交互，对于这样的交互，我们可能会需要重新创建已经在页面里用过一次的对象。

我们可以使用内置的Angular `$cacheFactory`服务来缓存不同的AWS对象，而不是进行复制。

18.16　`$cacheFactory`

`$cacheFactory`服务让我们能在需要的时候创建一个对象，或者在之前已经用过的情况下回收和重用一个对象。

要开始缓存，我们先要创建一个`dynamoCache`对象，在这里存储缓存的Dynamo对象：

```
// ...
self.$get = function($q, $cacheFactory) {
    var dynamoCache = $cacheFactory('dynamo'),
        credentialsDefer = $q.defer(),
        credentialsPromise = credentialsDefer.promise;

    return {
// ...
```

回到`dynamo`方法中，如果对象在缓存中存在的话，可以把它拖出来用，或者可以在需要的时候让它创建这个对象：

```
// ...
dynamo: function(params) {
    var d = $q.defer();
    credentialsPromise.then(function() {
        var table =
            dynamoCache.get(JSON.stringify(params));
        if (!table) {
            var table = new AWS.DynamoDB(params);
            dynamoCache.put(JSON.stringify(params), table);
       };
       d.resolve(table);
   });
   return d.promise;
},
// ...
```

18.17　保存 currentUser

用户登录和我们获取他的email时，就是一个把用户添加到我们用户数据库的好时机。

为了创建一个dynamo对象，我们需要再次使用promise API方法`.then()`，这次是在服务外。我们创建一个对象，它能让我们跟在Dynamo API控制台创建的用户表交互。

> 我们需要在首次运行应用时手动创建这些Dynamo表，因为直接给我们的Web用户创建dynamo表的权限是不安全的。

为了创建一个Dynamo表，我们要打开Dynamo控制台①，找到Create Table按钮。

我们想要创建一个叫做Users的表，其主键类型为Hash。Hash Property Name将会是一个主键，我们会用它在表上存取对象。我们会对这个示例程序使用字符串：User email，如图18-15所示。

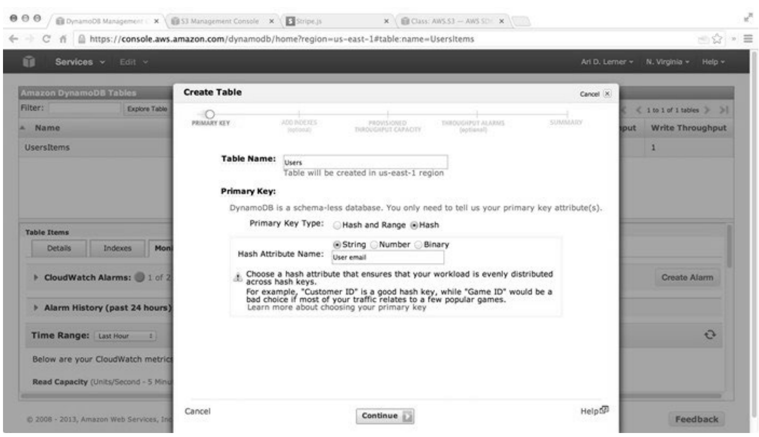


图18-15　创建Users Dynamo表

当我们点击到后面两屏时，会通过输入email的方式，建立一个基本的提醒。尽管这个步骤不是100%必要的，但很容易忘记我们的表还在创建中。如果没有提醒的话，可能就永远把它们忘在那里了。

① https://console.aws.amazon.com/dynamodb/home

我们点击到最终的review屏幕后，点击create，我们将会得到一个全新的Dynamo表，用户会被存储在那里。

当我们在控制台的时候，需要创建连接表。这个表把用户和用户上传的东西关联起来。

我们必须返回，再次找到Create Table按钮，创建一个叫做UsersItems的表，其主键类型为Hash and Range。对于这个表来说，Hash Attribute Name也是User email，Range Attribute Name则是ItemId。

用这种方式建立表，能让我们基于email查询所有创建了上传记录的用户。

接下来的一个屏幕上剩下的选项是可选的，可以点击过掉。

此时，我们就有了两个可用的dynamo表。

回到UserService，我们首先查询这个表，看看用户是不是已经存储在数据库中了；如果没有，就在我们的Dynamo数据库中创建一条记录。

```
var service = {
    _user: null,
    UsersTable: "Users",
    UserItemsTable: "UsersItems",
    // ...
    currentUser: function() {
        var d = $q.defer();
        if (service._user) {
            d.resolve(service._user);
       } else {
           // 加载了证书之后
           AWSService.credentials().then(function() {
               gapi.client.oauth2.userinfo.get()
                   .execute(function(e) {
               var email = e.email;
               // 为UsersTable
               // 为UsersTable获取dynamo实例
               AWSService.dynamo({
                   params: {TableName: service.UsersTable}
               })
               .then(function(table) {
                   // 根据email找到用户
                   table.getItem({
                       Key: {'User email': {S: email}}
                   }, function(err, data) {
                       if (Object.keys(data).length == 0) {
                           // 用户之前不存在
                           // 所以创建一条记录
                           var itemParams = {
                              Item: {
                                   'User email': {S: email},
                                  data: { S: JSON.stringify(e) }
                               }
                           };
                           table.putItem(itemParams,
                               function(err, data) {
                                   service._user = e;
                                   d.resolve(e);
                           });
                       } else {
                           // 用户已经存在
                           service._user =
```

```
                        JSON.parse(data.Item.data.S);
                    d.resolve(service._user);
                }
            });
        });
    });
  });
}
return d.promise;
},
// ...
```

尽管看起来代码很多，但它只是在我们的DynamoDB上作了一个基于用户名的查找或者创建。

此时，我们终于可以回到视图上看看发生了什么。

在templates/main.html文件中添加一个容器，它只是在用户不存在的时候显示登录状态，在用户已经存在时显示用户详情。

我们用简单的ng-show指令和新的google-signin指令来完成添加。

```
 <div class="container">
   <h1>Home</h1>
   <div ng-show="!user" class="row">
     <div class="col-md-12">
       <h2>Signup or login to ngroad</h2>
       <div google-signin
         client-id='395118764200'
         after-signin="signedIn(oauth)"></div>
     </div>
 </div>
  <div ng-show="user">
    <pre>{{ user | json }}</pre>
  </div>
</div>
```

视图设置好之后，可以在第二个<div>中跟已登录的用户互动（在生产中，把它做成单独的路由会比较好）。

18.18　上传到 S3

由于已登录用户已经存在于Dynamo中，是时候创建文件上传功能、直接把文件存到S3了。

首先，我们来简单了解一下跨域资源共享（Cross-Origin Resource Sharing，以下简称CORS）。它是一种现代浏览器支持的安全特性，允许我们使用一个标准协议来对不同域作出请求。

幸好，AWS团队已经把支持CORS变得非常简单。如果我们把自己的网站托管在S3上，甚至都不需要设置CORS（除了为了开发的目的）。

要在一个bucket上启用CORS，先要转到S3控制台[1]，并且找到我们要用来上传文件的bucket。对于这个示例，我们使用的是ng-newsletter-example bucket。

① https://console.aws.amazon.com/s3/home

定位到这个bucket之后，我们在它上面点击，加载Properties选项卡，展开Permissions选项。从这里，我们点击Add CORS configuration按钮，选择标准的CORS配置，如图18-16所示。

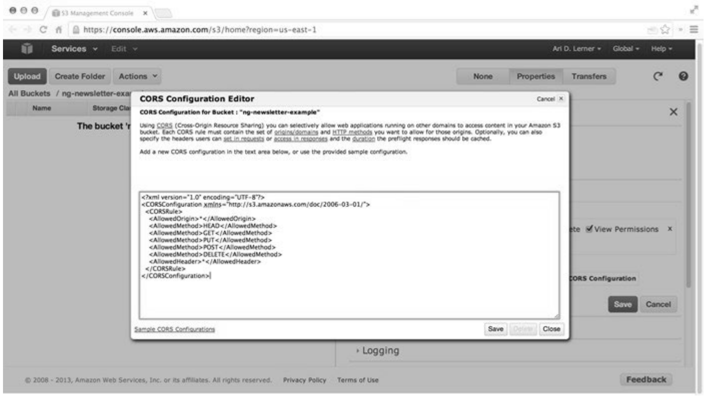


图18-16 在S3 bucket上启用CORS

我们要创建一个简单的文件上传指令，提供一个用HTML5 File API来处理文件上传的方法。这样，当用户选择一个文件时，文件上传会立即开始。

为了处理文件选择指令，我们创建一个简单的指令绑定到change事件上，并且在文件被选择之后调用一个方法。

这个指令很简单，如下所示：

```
 // ...
 .directive('fileUpload', function() {
     return {
         restrict: 'A',
         scope: { fileUpload: '&' },
         template: '<input type="file" id="file" /> ',
         replace: true,
         link: function(scope, ele, attrs) {
             ele.bind('change', function() {
                var file = ele[0].files;
                if (file) scope.fileUpload({files: file});
            })
         }
     }
})
```

我们可以这样在视图中使用这个指令，如下所示：

```
<!-- ... -->
<div class="row"
  <div class="col-md-12">
    <div file-upload="onFile(files)"></div>
  </div>
</div>
```

现在，当作出文件选择的时候，它就会调用当前作用域上的onFile(files)方法。

尽管在这里我们正在创建自己的文件指令，还是推荐查验一下ngUpload [https://github.com/twilson63/ngUpload]，以处理文件上传。

在onFile(files)方法中，我们要处理文件上传到S3，并且将记录保存到我们的Dynamo数据库表。我们想做Angular好公民，把这个功能放在UserService服务中，而不是放在控制器里。

首先，需要确认我们有这个能力获取一个S3 JavaScript对象，就像我们让dynamo可用那样。

```
// ...
var dynamoCache = $cacheFactory('dynamo'),
    s3Cache = $cacheFactory('s3Cache');
// ...
return {
    // ...
    s3: function(params) {
        var d = $q.defer();
        credentialsPromise.then(function() {
           var s3Obj = s3Cache.get(JSON.stringify(params));
           if (!s3Obj) {
               var s3Obj = new AWS.S3(params);
               s3Cache.put(JSON.stringify(params), s3Obj);
           }
           d.resolve(s3Obj);
       });
       return d.promise;
    },
// ...
```

这个方法与Dynamo对象创建的运行方式相同，它让我们可以直接对S3实例对象进行访问。我们很快就会看到这一点。

18.19　处理文件上传

为了处理文件上传，我们需要在UserService中创建一个叫做uploadItemForSale()的方法。出于规划的目的，就功能而言，我们想要：

- 将文件上传到S3；
- 为该文件获取一个signedUrl；
- 将这个信息保存到数据库。

我们将要在这个过程中使用当前用户，所以必须先确认这个用户存在，然后获取一个实例：

```
// 在scripts/services.js中
// ...
},
Bucket: 'ng-newsletter-example',
uploadItemForSale: function(items) {
    var d = $q.defer();
    service.currentUser().then(function(user) {
        // 处理上传
        AWSService.s3({
           params: {
               Bucket: service.Bucket
           }
       }).then(function(s3) {
           // 我们在S3对象中
           // 有一个s3 bucket句柄
       });
    });
```

18

```
        return d.promise;
    },
    // ...
```

S3 bucket的句柄让我们得以创建一个可供上传的文件。AWS在上传到S3的时候需要三个参数。

- Key：文件对象的键。
- Body：文件自身的blob。
- ContentType：文件类型。

所幸，当我们从浏览器获取文件对象的时候，它上面的信息都是可用的。

```
// ...
// 处理上传
AWSService.s3({
    params: {
        Bucket: service.Bucket
    }
}).then(function(s3) {
    // 我们在S3对象中
    // 有s3 bucket句柄
    var file = items[0]; // 获取第一个文件
    var params = {
        Key: file.name,
        Body: file,
        ContentType: file.type
    }

    s3.putObject(params, function(err, data) {
        // 文件已经上传
        // 或者上传过程中出错
    });
});
// ...
```

S3在上传文件时默认处于受保护状态。它什么都不用做，就可以防止我们把文件上传并且开放给公众。这个特性保证了我们上传到S3的所有东西都是受保护的，迫使我们在决定哪些文件要公开或者不要公开的问题上作出理智的选择。

记住这一点后，我们来创建一个临时地址，它会在一个给定时间之后失效。在ngroad市场中，这个URL会在供出售的每个东西上提供一个失效时间。

在任何情况下，要创建一个临时URL，我们都先要获取一个signedURL，并且把它存储在UsersItem关联表中：

```
// ...
s3.putObject(params, function(err, data) {
    if (!err) {
        var params = {
            Bucket: service.Bucket,
            Key: file.name,
            Expires: 900*4 // 1 hour
        };
        s3.getSignedUrl('getObject', params,
            function(err, url) {
                // 现在有了url
            });
    }
```

```
    });
});
// ...
```

最终，我们可以随上传的文件一起，把User对象保存在关联表中：

```
 // ...
 s3.getSignedUrl('getObject', params,
     function(err, url) {
         // 现在有了url
         AWSService.dynamo({
             params: {TableName: service.UserItemsTable}
         }).then(function(table) {
             var itemParams = {
                 Item: {
                    'ItemId': {S: file.name},
                    'User email': {S: user.email},
                    data: {
                        S: JSON.stringify({
                            itemId: file.name,
                            itemSize: file.size,
                            itemUrl: url
                        })
                    }
                }
            };
            table.putItem(itemParams, function(err, data) {
                d.resolve(data);
            });
        });
});
// ...
```

这个方法的完整版在：https://gist.github.com/auser/7316267#file-services-js-L98。

我们可以在控制器的`onFile`方法中使用这个新方法，编写的代码类似于：

```
$scope.onFile = function(files) {
    UserService.uploadItemForSale(files)
        .then(function(data) {
            // 刷新当前供出售的商品
        });
}
```

18.20　查询 Dynamo

在理想情况下，我们想要能列出某个用户能购买的所有产品。为了列出这些可能购买的物品，我们会使用query API。

Dynamo查询API有些深奥，乍一看相当混乱。

> Dynamo的文档：http://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_Query.html。

我们基本上会使用比较操作符来匹配对象模式，比如说`equal`、`lt`（小于）、`gt`（大于），或者其他任何更多的操作符。我们的连接表的键是User email键，所以我们要把这个键作为查询的

键，匹配到当前用户的email。

正如我们对属于用户的其他API做的那样，我们在UserService里面创建了一个方法，用于查询数据库：

```
 // ...
 itemsForSale: function() {
     var d = $q.defer();
     service.currentUser().then(function(user) {
         AWSService.dynamo({
             params: {TableName: service.UserItemsTable}
         }).then(function(table) {
             table.query({
                 TableName: service.UserItemsTable,
                KeyConditions: {
                    "User email": {
                        "ComparisonOperator": "EQ",
                        "AttributeValueList": [
                           {S: user.email}
                        ]
                    }
                }
            }, function(err, data) {
                var items = [];
                if (data) {
                    angular.forEach(data.Items, function(item) {
                        items.push(JSON.parse(item.data.S));
                    });
                    d.resolve(items);
                } else {
                    d.reject(err);
                }
            })
        });
    });
    return d.promise;
},
// ...
```

在上面的查询中，KeyConditions和“User email”都是必选参数。

18.21　在 HTML 显示列表

为了在HTML中显示用户的图像，我们只是把新的itemsForSale()方法的返回值赋给控制器作用域上的一个属性：

```
var getItemsForSale = function() {
    UserService.itemsForSale()
        .then(function(images) {
            $scope.images = images;
        });
    }
// 初始加载用户列表
getItemsForSale();
```

现在我们可以轻松地使用ng-repeat指令迭代出列表中的项，如图18-17所示。

```
<!-- ... -->
<div ng-show="images">
  <div class="col-sm-6 col-md-4"
    ng-repeat="image in images">
    <div class="thumbnail">
      <img ng-click="sellImage(image)"
        data-ng-src="{{image.itemUrl}}" />
    </div>
  </div>
</div>
```

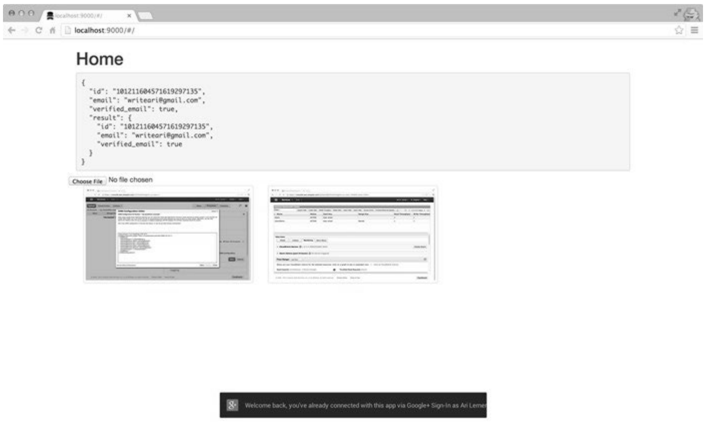


图18-17 图像清单

18.22 出售我们的作品

我们建立在AWS基础上的演示应用还差一步要做，就是从单页应用创建交易。

为了开始处理和接受支付，我们要创建一个`StripeService`，它为我们处理创建费用的工作。既然我们想要在模块的`.config()`方法中配置Stripe，那就需要创建一个`.provider()`。

这个服务自身是非常简单的，因为它把脏活累活都留给Stripe.js库去做了。

```
// ...
.provider('StripeService', function() {
    var self = this;

    self.setPublishableKey = function(key) {
        Stripe.setPublishableKey(key);
    }

    self.$get = function($q) {
      return {
          createCharge: function(obj) {
              var d = $q.defer();

              if (!obj.hasOwnProperty('number') ||
                  !obj.hasOwnProperty('cvc') ||
                  !obj.hasOwnProperty('exp_month') ||
                  !obj.hasOwnProperty('exp_year')
              ) {
                  d.reject("Bad input", obj);
```

```
                } else {
                    Stripe.card.createToken(obj,
                       function(status, resp) {
                            if (status == 200) {
                                d.resolve(resp);
                            } else {
                                d.reject(status);
                            }
                    });
                }

                return d.promise;
            }
        }
    }
});
```

如果你没有Stripe账号，那就到stripe.com[①]注册一个。Stripe是一个非常适合开发人员的支付处理网关，这使得它对于创建我们的ngroad市场来说非常理想。

一旦我们有了账号，就想找到Account Setting页面，定位到API Keys页面。我们的第一件事是要找到publishable key（可以是不真正产生费用的测试版，也可以是生产版本），并且把它记一下。

在我们的scripts/app.js文件中，我们只是添加了下面的代码，并且把“pk_test_YOUR_KEY”这个publishable key替换成了自己的。

```
// ...
.config(function(StripeServiceProvider) {
    StripeServiceProvider
        .setPublishableKey('pk_test_YOUR_KEY');
})
```

18.23　使用 Stripe

当用户点击他喜欢的图像时，我们在浏览器中打开一个表单，获取信用卡信息。我们要设置这个表单，让它提交到控制器上一个叫submitPayment()的操作。

注意到在上面有图像缩略图的地方，我们包含了一个操作，当图像被点击时，调用图像的sellImage()操作。

在MainController中实现sellImage()函数，如下所示：

```
// ...
$scope.sellImage = function(image) {
    $scope.showCC = true;
    $scope.currentItem = image;
}
// ...
```

现在，当图像被点击时，showCC属性会是true，我们可以显示信用卡表单。下面是一个极简表单：

```
 <div ng-show="showCC">
   <form ng-submit="submitPayment()">
```

① http://stripe.com

```
      <span ng-bind="errors"></span>
      <span>Card Number</span>
      <input type="text"
        ng-minlength="16"
        ng-maxlength="20"
        size="20"
        data-stripe="number"
       ng-model="charge.number" />
     <span>CVC</span>
     <input type="text"
       ng-minlength="3"
       ng-maxlength="4"
       data-stripe="cvc"
       ng-model="charge.cvc" />
     <span>Expiration (MM/YYYY)</span>
     <input type="text"
       ng-minlength="2"
       ng-maxlength="2"
       size="2"
       data-stripe="exp_month"
       ng-model="charge.exp_month" />
     <span> / </span>
     <input type="text"
       ng-minlength="4"
       ng-maxlength="4"
       size="4"
       data-stripe="exp-year"
       ng-model="charge.exp_year" />
     <input type="hidden"
       name="email"
       value="user.email" />
     <button type="submit">Submit Payment</button>
   </form>
 </div>
```

我们几乎是完全把作用域上的charge对象绑定到表单了，在创建费用时，会用到它。

这个表单自身提交到控制器作用域上的submitPayment()函数。submitPayment()函数如下所示：

```
// ...
$scope.submitPayment = function() {
    UserService
        .createPayment($scope.currentItem, $scope.charge)
        .then(function(data) {
            $scope.showCC = false;
    });
}
// ...
```

为了能接受费用，我们不得不做的最后一件事是实现UserService上的createPayment()方法。

现在，既然我们是在客户端做支付，从技术上讲是不可能处理支付的，只能接受stripeToken。可以设置一个后台进程来管理Stripe令牌到真实支付的转变。

在createPayment()函数中，我们需要调用StripeService来生成一个stripeToken。然后，我们把这个支付添加到一个Amazon SQS队列中，这样后台进程可以创建费用。

首先，我们使用AWS服务来访问SQS队列。

18

不像其他的服务，要让SQS服务运行，需要集成稍微多一点东西，因为这个服务要求我们有一个URL来跟它们交互。在我们的AWSService服务对象中，我们需要缓存正在处理的URL，并且改成每次使用生成的服务对象时创建一个新对象。不过，这个流程背后的原理还是完全一样的。

```
 // ...
 self.$get = function($q, $cacheFactory) {
     var dynamoCache = $cacheFactory('dynamo'),
         s3Cache = $cacheFactory('s3Cache'),
         sqsCache = $cacheFactory('sqs');
     // ...
     sqs: function(params) {
        var d = $q.defer();
        credentialsPromise.then(function() {
            var url = sqsCache.get(JSON.stringify(params)),
                queued = $q.defer();
            if (!url) {
                var sqs = new AWS.SQS();
                sqs.createQueue(params,
                    function(err, data) {
                        if (data) {
                            url = data.QueueUrl;
                            sqsCache.put(JSON.stringify(params), url);
                            queued.resolve(url);
                        } else {
                            queued.reject(err);
                        }
                });
            } else {
                queued.resolve(url);
            }
            queued.promise.then(function(url) {
                var queue =
                    new AWS.SQS({params: {QueueUrl: url}});
                d.resolve(queue);
            });
        })
        return d.promise;
    }
// ...
```

现在，我们可以在createPayment()函数中使用SQS了。SQS服务的一个注意事项是它只能发送简单信息，比如带有字符串和数字的信息。它不能发送对象，所以我们需要在任意想传送给队列的对象上调用JSON.stringify。

```
 // ...
 ChargeTable: "UserCharges",
 // ...
 createPayment: function(item, charge) {
     var d = $q.defer();
     StripeService.createCharge(charge)
         .then(function(data) {
             var stripeToken = data.id;
             AWSService.sqs(
                {QueueName: service.ChargeTable}
            ).then(function(queue) {
                queue.sendMessage({
                    MessageBody: JSON.stringify({
                        item: item,
```

```
                    stripeToken: stripeToken
                })
            }, function(err, data) {
                d.resolve(data);
            })
        })
    }, function(err) {
        d.reject(err);
    });
    return d.promise;
}
```

当我们提交表单的时候……如图18-18所示。

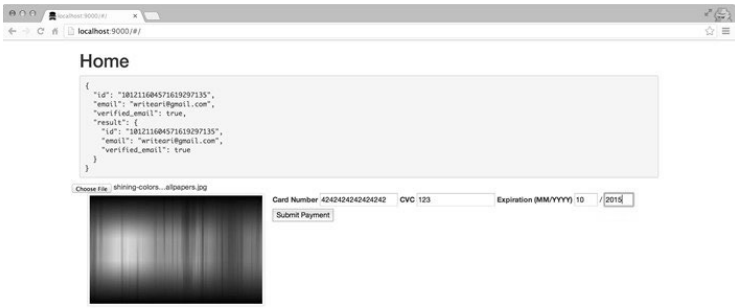


图18-18　处理支付

我们的SQS队列增长了，并且正好有一个支付等待完成，如图18-19所示。

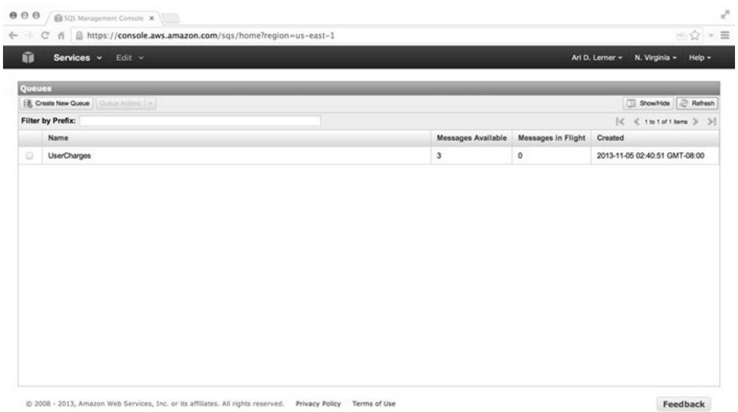


图18-19　SQS队列

18.24　使用 Firebase 的无服务器应用

作为一个客户端框架，单单Angular是不足以创建一个完整的后端应用的。通常很难知道什么时候要跟后端同步数据，怎么在修改内容的版本之间处理数据的变更和潜在冲突。

18

设想我们在同一时间有应用的两个实例在运行。如果两个实例都企图编辑相同的数据会怎样？不处理这种情况的话，就会出现问题，特别是如果我们创建的是一个复杂的Web应用前端，比如银行。

通过使用Firebase，我们能给Angular应用轻松地添加一个后端。在Angular.js主页上出现之后，Firebase很快就会成为Angular持久化的标准。

Firebase是一个实时后端，用于创建协同化的现代应用。Firebase让我们能够快速启动应用并将它运行起来，而不是要求我们忙着使用一个服务端组件来创建自定义的请求响应模型，还要在该模型中手动处理数据的同步。我们可以完全用Angular创建一个以数据为中心的Web应用，它能够开箱即用，实时更新所有的客户端。

存储在Firebase中的数据是标准的无模式JSON，这使得在Firebase中存储任意类型的数据模型非常容易。如果一个设备丢失了网络连接，Firebase继续允许对本地缓存数据的存取，并且当设备重新上线的时候，无缝地跟云端同步变更。

Firebase客户端库和REST API提供了从任何平台对数据的易访问性。尽管我们更关注的是Angular，这个事实意味着本地应用或者其他的服务端应用也可以访问Angular保存的数据。

当我们要对外发布应用的一个版本时，可使用Firebuse的一个托管服务，几秒钟之内将我们的Angular应用从命令行部署到我们自己的域。

18.25　使用 Firebase 和 Angular 的三方数据绑定

有了Angular，把内容渲染到浏览器是很容易的。Firebase是Angular一个出色的伙伴，因为它优雅地处理了数据的存取，这是产品级Web应用的另一个主要组件。

Angular的伟大之处在于，它在JavaScript模型和DOM之间的双向数据绑定。通过Angular模型与Firebase的同步，我们可以实时同步所有客户端上的应用模型。这意味着当一个客户端的数据变化时，这些变更立即保存到Firebase中，并且渲染到所有连接的设备上，如图18-20所示。

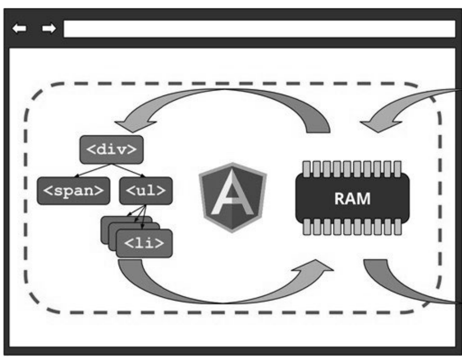


图18-20　客户端应用程序同步

当我们在这三个地方（视图，模型或者Firebase）中的任意一个里更新数据时，变更会实时传播到所有客户端的其他两个地方。

18.26　从 AngularFire 开始

得益于官方的Angular库，即AngularFire，使用Firebase和Angular创建实时Web应用很容易。Firebase团队专门为集成Angular应用创建了这个AngularFire绑定，我们将会看到这些。

使用AngularFire，要把我们的Angular应用后端建立到Firebase上只需四步。

18.26.1　注册并创建一个Firebase

在能够真正从Firebase中保存或者获取任意数据之前，我们需要一个账号。创建账号是免费的，所以我们来注册一下，如图18-21所示。

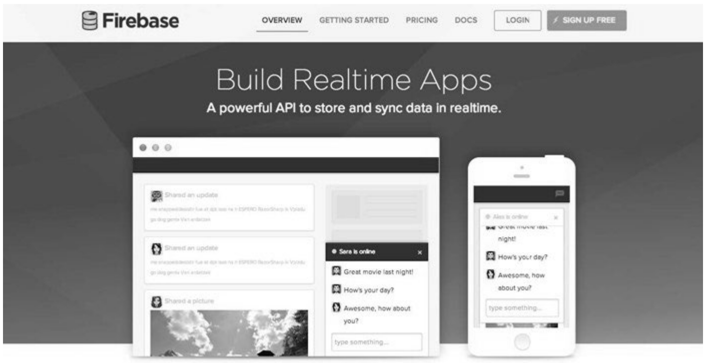


图18-21　注册过程

首先，进入firebase.com[①]，然后点击Sign up按钮（或者登录，如果有账号的话），如图18-22所示。

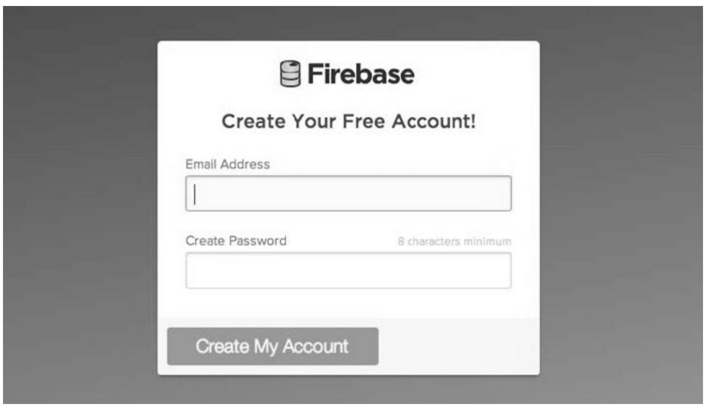


图18-22　注册过程

由于我们注册了一个账号，第一个Firebase已经自动为我们创建好了。它可以再Firebase面板上看到，如图18-23所示。

① http://firebase.com

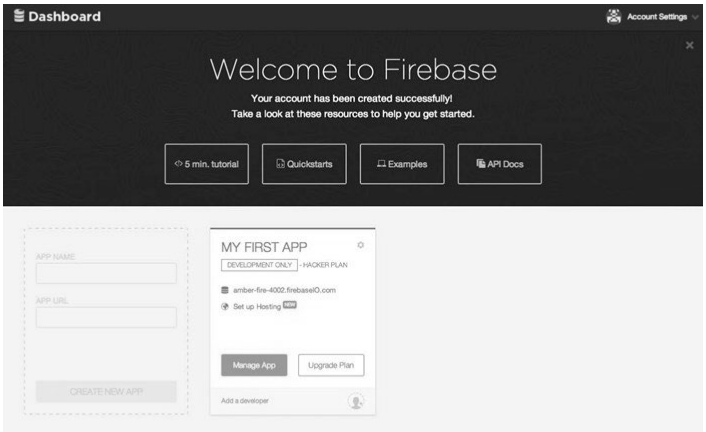


图18-23　注册过程

我们所选择的名字将会是用来指向Firebase数据的URL的一部分。例如，可以在URL https://ng-newsletter.firebaseio.com获得名为`ng-newsletter`的Firebase数据。

AngularFire绑定让我们把一个Firebase URL关联到一个模型，或者模型的集合。这些模型描述了一些数据，AngularFire会透明地在当前所有使用我们应用的客户端之间保持这些数据的同步。

Angular的双向绑定保持了DOM与内存中JavaScript变量的同步，Firebase保存了这些变更，并且实时把它们发送给所有监听的客户端。

我们没有改变创建Angular应用的方式，就获得了这种数据同步机制，太酷了。

18.26.2　包含Firebase和AngularFire库

使用AngulaFire也是很容易的，只需将两个JavaScript文件包含到我们的HTML文件中：一个是Firebase，另一个是AngularFire。

我们需要使用Firebase的CDN，所以在index.html的顶部，我们添加下面两行代码：

```
<script
  src="https://cdn.firebase.com/v0/firebase.js"></script>
<script
  src="https://cdn.firebase.com/libs/angularfire/0.5.0/angularfire.js">
</script>
```

18.26.3　把Firebase作为依赖项添加

像平时对待任何应用库一样，我们需要把Firebase库设置为模块的一个依赖项。这会告诉应用的其他部分，我们可以在应用中使用Firebase绑定：

```
angular.module("myapp", ["firebase"]);
```

18.26.4　绑定模型到Firebase URL

通过把Firebase定义为依赖项的方式，我们现在可以获得对`$firebase`服务的访问，这能让我

们把它作为依赖项注入到控制器和服务中。

```
angular.module('myapp', ['firebase'])
    .controller("MyController", ["$scope", "$firebase",
        function($scope, $firebase) {
            // 把控制器的定义写在这里
    }
]);
```

$firebase服务带有一个参数：Firebase引用。

FirebaseRef（Firebase引用）

Firebase引用告诉$firebase数据存在哪里，如何连接。$firebase服务处理与Angular的同步，并且是我们调用方法保存变更的地方。

这个对象有多个方法，我们可以用来与远程数据交互。这些方法在下面详细列出，都是以$符号开头的（例如，$add()、$save()），都可以在这个对象上使用。

> 注意，对象的变更不会引起远程数据的变更。

为了把本地对象模型同步到远程的Firebase引用，我们使用服务的方法，并且传入一个Firebase对象的实例。例如，要把$scope.items模型同步到我们的ng-newsletter项，运行如下方法：

```
angular.module('myApp')
.controller("MyController", function($scope, $firebase) {
    // Firebase URL
    var URL = "https://ng-newsletter.firebase.com";
    // 同步$scope上的items
    $scope.items = $firebase(new Firebase(URL + '/items'));
 });
```

至此，我们可以简单地跟$scope.items对象交互，这样可以同步我们的Angular模型和Firebase。

18.26.5　数据同步

我们可以使用下面$firebase对象提供的这些方法来把数据同步到Firebase。

1. **$add(value)**

$add方法带有单个任意类型的参数。它把这个值添加为一个按照时间排序的列表成员。我们可以把这个看作是在Firebase引用数组上调用.push(value)。

> 注意，Firebase引用对象并不真的是个数组，但是可以把它当成像数组那样用。

例如，我们可以在Firebase引用的/foo端点上添加一个字符串“bar”：

```
$scope.items.$add({foo: "bar"});
```

2. **$remove(key)**

$remove方法从Firebase上移除远程的子引用。它带有单个可选参数。

key（可选，字符串类型） 如果我们提供了一个key参数（作为字符串），$remove()方法会移除这个key指向的子对象。如果没有提供key，它就会移除整个远程对象。

```
$scope.items.$remove("foo"); // 移除名为"foo"的子对象
$scope.items.$remove(); // 移除整个对象
```

3. $save(key)

$save方法使用Firebase数据存储同步本地元素上的所有变更，并且立即把它们推送到所有监听的客户端。它带有一个参数。

key（可选，字符串类型）如果我们提供了key参数（字符串），$save方法会把对key指向的子元素的变更保存到Firebase。如果没有给$save()方法提供key，所有对这个对象的本地变更都会保存到Firebase。

> $save()方法最常用于保存本地变更的模型。

```
$scope.items.foo = "baz";
$scope.items.$save("foo"); // 新的Firebase(URL + "/foo")现在包含了"baz"
```

4. $child(key)

$chile()方法为给定key指向的子对象创建了一个新的$firebase对象。这个方法带有单个参数：

key（字符串类型）key字符串被用于指向新创建的子对象。

```
var child = $scope.items.$child("foo");
// 等同于调用
// $scope.items.$remove("foo");
child.$remove();
```

5. $set(value)

$set()方法把这个对象的远程值覆盖为新值。$set()方法也会将本地对象的版本更新为这个值。

它带有一个参数：

value（对象类型）value参数是本地对象的新值。value覆盖了旧值，随后更新到这个新值。

```
$scope.items.$set({bar: "baz"}); // 新的Firebase(URL + "/foo")现在是null
```

18.27　在 AngularFire 中排序

如果我们想要对远程对象排序，可以在调用$save()之前设置一个记录上的$priority字段，而不是简单地在本地用Angular的orderBy过滤器排序。

```
$scope.items.foo.$priority = 2;
$scope.items.$save("foo"); // 新的Firebase(URL + "foo")的优先级现在是2
```

$firebase服务默认返回一个简单的JavaScript对象。我们可以把这个对象转换为一个数组，简单地使用orderByPriority过滤器来排序。

这个过滤器把$firebase服务返回的对象转换成一个数组，根据Firebase定义的优先级来排

序。AngularFire会在每个对象上设置一个$id属性，它引用了对象的keyname。

```
<ul ng-repeat="item in items | orderByPriority">
  <li>
    <input type="text" id="{{item.$id}}" ng-model="item.$priority"/>
    {{item.name}}
  </li>
</ul>
```

18.28 Firebase 事件

Firebase触发两类事件，我们可以在应用内用这两类事件来处理自定义逻辑。我们可以使用$on()方法来给这两种事件类型添加事件处理程序。

1. loaded

当从Firebase收到初始数据，从一个连接初始化的时候，Firebase触发loaded事件。它会且只会被触发一次。

```
$scope.items.$on('loaded', function() {
    console.log("Items loaded");
});
```

2. change

每当有一个远程的变化数据应用于本地对象时，Firebase都会触发change事件。例如，如果有另外一个用户往我们的任务列表里加了个新任务，它就会触发，如下所示：

```
$scope.items.$on('change', function() {
    console.log("A change is afoot");
});
```

18.29 显式同步

要给一个$scope变量添加自动化的显式同步机制，可以调用$firebase服务返回对象的$bind()方法。

$bind()方法自动建立了一个三向绑定，所以我们不用显式使用$add()或者$save方法在Firebase上保存数据。

```
$scope.items.$bind($scope, "remoteItems");
$scope.remoteItems.bar = "foo"; //新的Firebase(URL + "/bar")现在是"foo"
```

$bind()方法返回了一个promise，它会从收到服务器的初始数据之后执行。这个promise会被用一个unbind方法执行，unbind在调用时可以解除三向绑定。

```
$scope.items.$bind($scope, "remote")
.then(function(unbind) {
unbind();
    // 远程数据
    // 没有产生变更
    $scope.remote.bar = "foo";
 });
```

18

`$bind()`方法返回了一个promise，它会在AngularFire从收到服务器的初始数据之后执行。这个promise会被用一个unbind方法执行，unbind在被调用时，可以解除三向绑定。这个关联的解除对于优化网站和移除不必要的监控是很有帮助的。

18.30　用 AngularFire 进行认证

Firebase提供了一个简单的开箱即用的客户端认证策略。

使用Firebase的Simple Login或者Custom Login方法，我们可以轻松地用AngularFire给应用添加用户认证。

如果我们有自己的服务器，想要控制自己的认证过程，或者我们想把自己的认证过程跟Firebase集成，Custom Login是最适合用的。

如果我们想用Firebase来管理我们所有的认证，可以使用Simple Login，它支持Facebook、Twitter、Github、Persona和Email/Password认证。

通过在应用模块中定义Firebase为依赖项的方式，我们在控制器和服务中获得了对`$firebaseAuth`服务的访问权。

```
angular.module('myApp')
.controller("MyAuthController", function($scope, $firebaseAuth) {
    // 在这里定义我们的控制器
});
```

`$firebaseAuth`服务方法带有两个参数：一个Firebase引用和一个可选的选项对象。为了能自定义使用Firebase进行认证的方式，这个对象可以包含如下属性。

- `path`：如果`authRequired`属性在`$routeProvider`中被设置成`true`，用户又没有登录的话，用户会被重定向到`path`属性指向的地址。
- `simple`：`$firebaseAuth`默认需要包含firebase-simple-login.js文件，如果`simple`的值被设置为`false`，就没这个要求了，但只有自定义登录功能会被启用，我们不能使用简单认证了。
- `callback`：当认证状态发生变化的时候，Firebase会调用这个函数。我们可以把这个回调作为在`$rootScope`上触发的事件的替代，这是处理认证状态变更的推荐方式。

```
angular.module('myApp')
.controller("MyAuthController", function($scope, $firebaseAuth) {
    var ref = new Firebase(URL);
    $scope.auth = $firebaseAuth(ref);
    // $scope.auth.user在用户登录之前都是null
});
```

`$firebaseAuth()`方法返回的对象包含一个叫做`user`的属性。如果用户注销了，`user`就被设置为`null`，一旦他登录了，`user`变成包含用户详情的一个对象。我们会在下面探讨登录的检测。

> 用户详情对象的内容会随着所使用认证方式的不同而有所区别，但至少，它会有一个user id和provieder name。

18.31 认证事件

使用AngularFire认证，我们能够使用多个可改变用户认证状态的方法，这些方法是$login()、$logout()和$createUser()。

在AngularFire中，认证状态被认为是全局状态，下面的每个认证方法都会被在$rootScope上广播。既然几乎所有作用域都是从$rootScope继承的，我们可以从任意控制器调用$scope.on(...)。

全局认证意味着不会有多个用户同时登录到应用的同一个实例。例如，全局认证阻止了两个用户在同一个浏览器实例中同时登录到Gmail。

$firebaseAuth:login 用户成功登陆时会触发这个事件。它会用两个参数来触发：event和user对象。

```
$rootScope.$on("$firebaseAuth:login", function(evt, user) {
    console.log("User " + user.id + " successfully logged in!");
});
```

$firebaseAuth:logout 用户注销时触发这个logout事件。这个事件使用一个event参数来触发。

```
$rootScope.$on("$firebaseAuth:logout", function(evt) {
    console.log("User logged out!");
});
```

$firebaseAuth:error 当调用$login()或者$logout()的过程中产生错误时，error事件会触发。这个事件使用一个error参数来触发。

$login(token, [options]) 我们使用$login()方法来登录一个用户。通常在用户点击登录按钮时使用它，如下所示：

```
<a href="#"
  ng-hide="auth.user"
  ng-click="auth.$login('persona')">Login</a>
```

$login()函数最多可带两个参数：

tokenOrProvider (string/JWT token) 如果我们正在使用Firebase Simple Login，可以简单地传入一个提供者名称，比如facebook或者persona。如果我们想用使用Custom Login流程，就需要传入一个合法的JWT令牌了。

options (object) 我们只在使用Simple Login时用到这个options参数，提供的这个options会不经修改地传递给Simple Login方法。

对于password提供者来说，我们会需要把username和password作为对象提供。

更多关于user对象的信息，参阅AngularFire.com[1]的Firebase文档。

① http://angularfire.com/

18.31.1　$logout()

$logout()方法注销当前用户，不带参数。

$firebaseAuth:logout事件会在注销结束之后触发，将user属性设置为null。我们一般会把这个方法添加在注销按钮上：

```
<span ng-show="auth.user">
  {{auth.user.name}} | <a href="#" ng-click="auth.$logout()">Logout</a>
</span>
```

18.31.2　$createUser()

当我们使用Firebase Simple Login的“password”提供者时，$createUser()很有用。

$createUser()方法带三个参数。

email (string)　我们要用email来创建用户

password (string)　我们要用password来创建用户

callback (function)　Firebase在$createUser()运行完之后，调用这个callback方法。它带有两个参数：error和user。如果在$createUser()方法中产生了错误，error就会包含错误信息，user就会是null。如果error是null，那user就会被定义。

```
auth.createUser(email, password, function(error, user) {
    if (!error) {
        console.log('User Id: ' + user.id + ', Email: ' + user.email);
    }
});
```

Firebase使我们的Angular应用接通一个后端变得很容易，无需担心要建立一个服务器或者写一行后端代码。AngularFire使得我们能创建复杂、实时的应用，这些应用能立即在应用的模型和Firebase中存储的数据之间同步。

要对AngularFire了解更多，源码可以在Github上找到。

想在几分钟内把一个AngularFire应用运行起来，那就将angularFire-seed[1]仓库复制一下。

18.32　使用 Firebase 托管部署你的 Angular 应用

现在已经有一个正在工作的Angular应用了，如果有一种很简单的方式部署和托管它不是很好吗？Firebase有一个叫做Firebase托管的服务，它允许你使用安全的SSL连接和CDN服务托管静态内容。为了开始托管，在你的Firebase面板上点击“设置托管”链接。通过3步即可让你的应用运行在你自己的firebaseapp.com上！

18.32.1　安装Firebase工具

要安装Firebase命令行工具，简单地运行npm install -g firebase-tools命令即可。注意，

① https://github.com/firebase/angularFire-seed

你需要先安装Node后才能使用这个命令。

18.32.2　部署你的Web站点

在命令行中，使用cd命令进入应用目录，运行`firebase init`命令初始化你的应用。然后运行`firebase deploy`命令。现在你的应用就运行了！在Firebase面板中，你可以点击回滚到之前的部署。

使用Firebase的Hacker Paln，可以将应用部署到firebaseapp.com上。如果你想部署到你自己的自定义域中，可以使用Firebase的付费方案做到这一点。

18.33　除了 AngularFire 之外

AngularFire是用于跟Firebase交互的一个很好的封装器，但是要从Angular做更复杂的操作，当然也可以直接用Firebase SDK。要了解更多关于这个复杂的实时平台的高级功能，请参阅Firebase教程[①]。

① https://www.firebase.com/tutorial/

第 19 章

测　　试

Angular框架鼓励编写干净、可靠、可测试的代码。这是Angular带来的最有价值的特性之一。

Angular团队非常强调测试的重要性，他们创建了一个测试运行器来让这个过程更简单。他们表示：

> JavaScript是一种动态语言，有强大的表达能力，但带来的问题是：从编辑器那边得不到什么帮助。因为这个原因，我们强烈感觉到：任何使用JavaScript编写的代码都应当有强大的测试集。我们已经在Angular里面加了不少特性，能让测试Angular应用非常容易。所以没有理由不做测试。

19.1　为什么要做测试

不管是为了何种商业目的，对代码的信心是很重要的。当代码库有了测试的支撑，就可以了解我们代码的各部分是否按预期工作。

代码中的bug是不可避免的，没有测试，很难知道它们藏在哪里；测试能够分离和消除这些缺陷。这可以让其他开发人员容易上手，并且提供代码的可用文档。

如果我们想要了解在应用中发生了什么事情，测试是至关重要的。

19.2　测试策略

在开发Angular应用的测试套件时，对于如何在应用中测试、要测试什么，能够有所规划，总是好事。如果最终没有验证过实际的东西，写着毫无意义的测试，我们对于应用是否能正常运行不会有信心。相反，如果能测试所想到的一切，最终，我们花在编写测试和找出测试代码中微小bug的时间，比花在应用上的时间要多。

能从所编写的测试中得到什么价值，需要测试什么，对此务实一些很重要。

最终，测试既是一种衡量我们应用健康度的工具，也是一种度量，当引入新功能的时候，它能告诉我们代码是否出问题了。

19.3　开始测试

在开始测试之前，一个最大障碍是要建立一个测试的运行器来运行我们代码的测试。

JavaScript代码的测试也是有些困难的，因为这要求我们把自动化功能放在浏览器里。

构建一个开发测试套件已经够困难了，想要支持持续集成会怎样呢？这样我们新部署的代码可以被自动测试，也可以在创建新版本之前对代码质量有信心。

> 在软件工程中，持续集成是这样的实践：一天内多次合并共享同一主线的开发工作的副本，并且在更新的基础上运行测试套件。

Karma[①]是一个测试工具，它从头开始构建，免去了设置测试方面的负担，这样我们就可以将主要精力放在构建核心应用逻辑上。

Karma产生一个浏览器实例（或者多个不同的浏览器实例），针对不同的浏览器实例运行测试，检测在不同浏览器环境下测试是否通过。Karma与浏览器通过`socket.io`来联系，这能让Karma保持持续通信。因此Karma提供了关于哪些测试正在运行的实时反馈，提供一份适合人类阅读的输出，告诉我们哪些测试通过、哪些失败或者超时。

Karma可以原生地与多个不同浏览器通信，消除了手工在多个浏览器上测试代码的需要。比如说，它可以在Chrome、IE、FireFox上运行测试，把结果分别输出到控制台。我们甚至可以连接到自己的本地设备（是的，比如iPhone或者iPad）来测试代码。

19.4 AngularJS 测试的类型

要测试我们的Angular应用，有好几种不同的方式，取决于我们想关注什么级别的粒度，想要完成什么功能。

19.4.1 单元测试

我们可以专注于构建我们的测试来隔离具体、单独组件的代码。这种方法被称为“单元测试”，在不同阶段、不同条件下、以不同的输入来测试各种特定单元的代码。

单元测试专门用于测试小型、独立的代码单元，单个函数，或者较小的带有功能的交互。它不是用来测试大功能集的。

单元测试的麻烦在于把逻辑隔离成小块，这样我们才能测试它。本章的后面部分会讨论实现隔离的策略。

何时选择单元测试

当编写功能代码的时候，我们将会创建小的功能组件。例如，在构建一个应用用于实时过滤列表中元素时，会构建一个过滤器功能。

这个过滤器的功能是一个功能单元，当我们需要使用单元测试进行测试时，它是理想的选择。要确保这个功能已经实现，按照预期工作，我们需要隔离这个组件，并且用不同的输入来测试。

> 设想我们正在构建一艘火箭飞船。我们想要测试每艘飞船的一部分（例如推进器、操纵杆控制、供氧系统）来验证飞船通常是按照我们期望的那样工作的。

① http://karma-runner.github.io/0.10/index.html

19.4.2　端到端测试

另一方面，也可以对我们的应用作黑盒测试（也就是端到端测试）。在端到端（E2E）测试中，我们测试应用的视角是：作为最终用户，对系统的底层实现一无所知。这种方法非常适合测试大型应用的功能。

端到端测试适合测试页面上的用户交互，无需手动刷新页面。

关于这类测试的介绍并不新鲜，还有些了不起的工具，能使我们建立自动化的浏览器测试。我们可以使用工具如PhantomJS或CasperJS来进行无头浏览器测试（即不必打开浏览器），或Karma等工具，会真正将打开一个浏览器并在一个iframe中执行所有的测试。

何时选择端到端测试

当我们编写用例功能的测试时，顺着用户的思路去写测试总是不错的。端到端测试意义重大，因为它映射了用户在使用我们应用时的真实体验。

例如，当建立一个用户登录流程时，我们要测试用户登录进来并且重定向到了他的主页。我们并不关心用户是怎样登录进来的，我们只关心他们登进来，然后跳转到恰当的位置。

> 想象一下你正在构建一个火箭船。端到端测试不在乎发动机或起落架，它只关心火箭起飞，把你的宇航员送上太空。

Karma测试运行器同时支持单元测试和端到端测试。

> 注意，编写单元测试而不是端到端测试将使我们的测试速度飞快。把测试设置成同步执行，使用模拟库，也将大大加速我们的测试。

19.5　开始

要运行我们的测试，需要安装Karma测试运行器。读到这的时候，你应该已经装了NodeJS[1]和npm了吧，要是还没有，快去装吧。安装好之后，我们就可以用npm命令来安装Karma了：

```
$ npm install -g karma
```

> 我们将把依赖关系保存在package.json文件中。如果要在npm已安装的情况下设置package.json，直接运行`npm init`，跟着向导走一遍就行了。

要开始测试我们的应用，需要为应用代码和测试代码设置一个合理的结构。

推荐用下面的格式存储应用的文件：

```
app/
  index.html
  js/
    app.js
    controllers.js
```

① http://nodejs.org

```
      directives.js
      services.js
      filters.js
    views/
      home.html
      dashboard.html
      calendar.html
  test/
    karma-e2e.conf.js
    karma.conf.js
    lib/
      angular-mocks.js
      helpers.js
    unit/
    e2e/
```

app/的布局是标准的，应用代码划分之后存储在里面。test/目录中嵌套存放了测试，放置在对应的目录中，目录名称反映了测试的类型：unit/或者e2e/。

> 当前版本Angular的文件类型最佳结构尚有争议，这是一种推荐的测试文件结构的布局。

在test/目录中，有两个不同类型的Karma配置文件。每个文件都包含了将要运行的具体测试类型。随着对每种测试类型的遍历，我们会讨论每种Karma配置文件应该长什么样，以及如何为我们的用途来定制它。

运行一个Karma测试挺容易的：`karma start path/to/karma.config.js`。当测试运行器启动的时候，它会把在Karma配置文件中列出的浏览器启动起来，如图19-1所示。

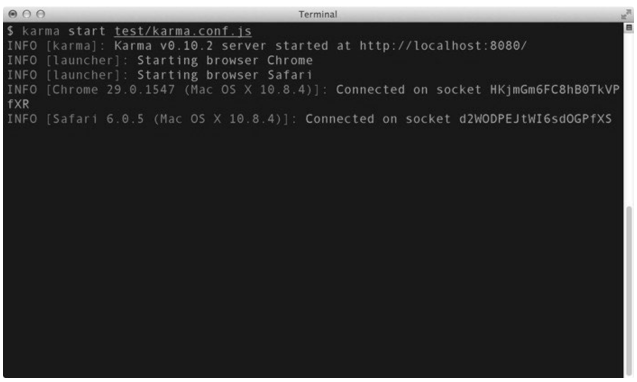


图19-1　用Chrome和Safari运行Karma

默认情况下，如果不是另有规定，Karma将监控配置文件中列出的所有文件。任何时候有文件产生了变化，Karma都将运行适当的测试。

19.6　初始化 Karma 配置文件

Karma给了我们一个生成器来创建配置文件。这个生成器会问几个关于要怎样建立配置文件的问题，每个问题建议了一个默认值，可以简单地接受所有默认值，过一会我们就来这么做，如

图19-2所示。

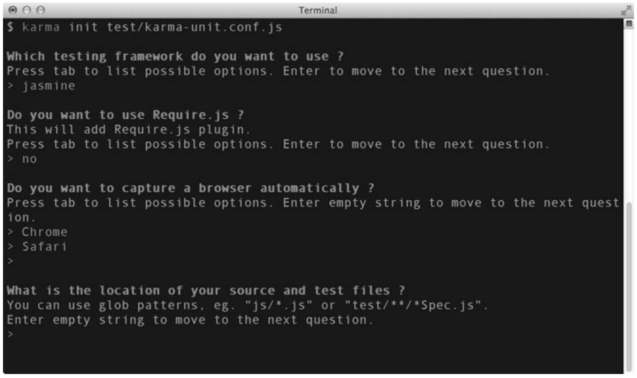


图19-2　Karma初始化

设置用单元测试和端到端测试来做测试的过程大体相同。我们会使用`karma init`生成器来创建karma.conf.js配置文件。

1. 建立单元测试

首先，让我们在测试文件的路径运行`karma init`命令，在本例中，我们在tests目录中创建Karma配置文件：

```
$ karma init test/karma.conf.js
```

对于单元测试而言，运行测试所需的依赖项都要具备。当使用Karma生成器来构建单元测试时，单元测试包含下列这些代码的引用是很重要的：

- 一个测试框架（选一个）
  - Jasmine（默认）
  - Mocha
  - QUnit
- 自定义的配置文件（需要w/Mocha）
- 所需的第三方代码
- 应用特有的代码
- 测试代码
- 模拟用的angular-mock.js库

单元测试需要引用待测试应用的所有代码，也要引用所有要写的测试代码。

例如，一个示例单元测试的Karma配置文件可能看起来像下面这样（为了简单起见，去掉了注释）：

```
module.exports = function(config) {
  config.set({
    basePath: '..',
    frameworks: ['jasmine'],
    files: [
      'lib/angular.js',
```

```
      'lib/angular-route.js',
      'test/lib/angular-mocks.js',
      'js/**/*.js',
      'test/unit/**/*.js'
    ],
    exclude: [],
    port: 8080,
    logLevel: config.LOG_INFO,
    autoWatch: true,
    browsers: ['Safari'],
    singleRun: false
  });
};
```

这个配置文件类似于我们会生成出来的那样。

设置了这个文件之后，就能像下面这样运行单元测试了：

```
$ karma run test/karma.conf.js
```

另外，如果想在每次代码有变化时运行单元测试（如果把autoWatch设置成了true），可以像下面这样：

```
$ karma start test/karma.conf.js
```

2. 建立端到端测试

要设置端到端测试，要使用端到端测试Karma配置文件的路径来运行Karma生成器。

```
$ karma init test/karma-e2e.conf.js
```

端到端测试要使用ng-scenario框架。不像单元测试那样，我们不需要引用所有的库代码：端到端测试是跑在服务器上的，ng-scenario框架只需要在浏览器中加载所有这些测试就可以了。

端到端测试的示例Karma配置文件可能长这样：

```
module.exports = function(config) {
  config.set({
    basePath: '..',
    frameworks: ['ng-scenario'],
    files: [
      'test/e2e/**/*.js'
    ],
    exclude: [],
    port: 8080,
    logLevel: config.LOG_INFO,
    autoWatch: false,
    browsers: ['Chrome'],
    singleRun: false,
    urlRoot: '/_karma_/',
    proxies: {
      '/': 'http://localhost:9000/'
    }
  });
};
```

设置好这个配置后，就可以这样运行端到端测试：

```
$ karma run test/karma-e2e.conf.js
```

另外，如果想要在每次代码有变化时运行我们的测试（如果把autoWatch设置成了true），可以像下面这样：

```
$ karma start test/karma-e2e.conf.js
```

19.7　配置选项

基于Karma，可以在多种配置选项之间选择，定制自己喜欢的测试方式。

1. 框架

生成器会问我们使用哪个测试框架来做测试。Jasmine是默认的测试框架，不过，生成器默认也支持Mocha、QUnit和其他测试框架。

这些测试框架都需要安装额外的npm库。例如，要使用Jasmine框架，需要安装Jasmine插件。

```
$ npm install --save-dev karma-jasmine
```

> 使用--save-dev标志会把依赖关系写到package.json文件中，放在devDependencies下面。

在配置文件里，这个标志使用了一个数组，可以让我们使用多个框架。一般我们只用一个，所以通常这个选项会被设置成['jasmine']或者['mocha']。

比如：

```
frameworks: ['jasmine'],
```

2. RequireJS

如果这个项目使用了RequireJS[1]库，询问是否要包含RequireJS的，就要回答yes。如果项目没有包含它，不是把项目中所有文件都列在Karma配置文件中（马上就会看到），而是要包含单独的测试文件，它负责加载特定模块。

RequireJS是专门为浏览器设计的JavaScript文件和模块加载器。利用它我们能编写JavaScript库，这些库能够导出一个库并使用模块名来配置依赖的预期，它将在我们模块加载的时候可用。

它的主要好处是：

- 建立了一个导入过程；
- 能加载嵌套的依赖项；
- 让打包依赖项变得容易。

实际上，RequireJS允许我们通过模块来定义JavaScript，以及在JavaScript中请求这些模块。例如：

```
define(['jquery', 'underscopre'],
  function($, _) {
    // $ 引用jQuery
```

① http://requirejs.org

```
    // _ 引用underscore
});
```

有关如何设置测试的更多信息，参见RequireJS。

3. 浏览器捕获

Karma生成器会询问要自动启动哪个浏览器来捕获测试结果。终止测试运行器时，Karma也会把这些浏览器关掉。我们也可以用浏览器打开Karma Web服务器监听的URL（默认为http://localhost:9876）来测试，如果要从本地网络的另一台机器（或者虚拟机）的IE来测试的话，这一点值得牢记。

要使用Karma来启动和运行，每个浏览器都需要安装额外的插件。我们可以使用npm来安装这些插件。例如，为了让Karma控制Chrome，我们就要安装Chrome启动插件。

```
$ npm install --save-dev karma-chrome-launcher
```

如果要使用Safari，就要安装Safari启动插件，对于Firefox，就要安装Firefox启动插件，其他类同。

```
browsers: ['Chrome', 'Safari'],
```

4. 源文件和测试文件

Karma生成器会询问JavaScript源文件和测试文件存放在哪里，这个数组可以包含简单的字符串和对象。

字符串可以是模式（例如app/js/**/*.js），或者文件地址（例如app/js/main.js），这些文件和模式都是相对于basePath的。

我们也可以用一个对象（而不是字符串）来指定文件，要配置一个给定的文件路径或者模式的某些方面时，这是比较有用的，在下面的例子里，这个对象告诉Karma监控文件public/js/watch-me.js的变更，但不在页面上包含它，也不把它提供成URL：

```
{
    pattern: 'public/js/watch-me.js',
    watched: true,
    included: false,
    served: false
}
```

注意，使用对象的原因是要对文件或者文件模式提供细粒度的控制。因此，pattern属性是必须的。其他属性，比如included，是有默认值的，所以只有当模式偏离常态的时候才需要设置。

我们来讨论一下每个选项和它们默认值的细节。

pattern 模式是一个正则表达式，匹配测试文件，这个选项可以是单个文件，也可以是文件的一种模式，它们符合前面列出的字符串的特征。

watched 如果Karma设置成使用autoWatch，这个布尔值用于指定文件是否会被监控。如果它被列为true，Karma会在这个文件被修改时运行测试。如果设置成false，这些测试就不会运行。

如果watched没有列在配置对象的属性中，这个对象列出的文件默认会被监控（true）。

included　这个布尔值告诉Karma在浏览器中使用<script>标签来加载文件。如果这个选项设置成true，浏览器会加载这些文件。如果设置成false，我们就要负责手动加载它们。一般这个选项是跟RequireJS 结合使用的。

默认情况下，文件被设置成使用<script>标签包含（true）。

served　这个布尔值告诉Karma通过Karma Web服务器提供这个文件。如果设置成true，这个文件就可以通过Web服务器来访问，如果是false，就不能。默认这个选项设置成true，所以这些文件能在Web服务器上访问。

5. 顺序

文件列出的顺序很重要，所以我们要把库放在应用文件之前，因为它们会被依赖。如果列了一个模式，文件会按照字母顺序排序然后包含进来。

每个文件只会被包含一次，所以如果一个文件被匹配了多个模式，它只会被包含一次。

关于files属性的完整示例：

```
files: [
  // 简单字符串，可用于定位单个文件
  'js/app/vendor/angular/angular.js',
  // 或者可以用模式来定位一堆文件
  'js/app/*.js',
  // 对象
  // 当index.html文件变更时
  // 不运行测试
  pattern: 'public/index.html', watched: false},
  // 我们也可以设置文件不被包含但仍然受监控
  {pattern: 'public/index.html', included: false}
]
```

6. exclude

Karma可以排除那些我们在加载测试时不想引入的文件。利用exclude选项就可以设置一个不想默认包含的文件列表。比如说，如果你使用了RequireJS，这个数据就很有用了。

```
exclude: [
  'public/index.html'
]
```

7. basePath

使用basePath选项，可以把根路径地址设置为在files和exclude属性中定义的相对路径。如果basePath选项是一个相对地址，它会被解析成相对于Karma配置文件的位置（__dirname）。

```
basePath: '..',
```

8. autoWatch

将autoWatch属性设置为true会让karma在files中的文件发生变更时执行已配置的测试。使用持续集成服务器时，监控文件变化是不必要的，这时把这个设置成false就很有用了。

```
autoWatch: true,
```

9. captureTimeout

如果浏览器加载时间超过captureTimeout（默认是60秒或者60 000毫秒），Karma会把进程杀掉，再试一次。如果它试了三次还是失败，Karma就不再尝试启动浏览器了。

```
captureTimeout: 60000
```

10. colors

Karma的默认输出是有颜色的。如果不想在终端中显示带颜色的输出，可以把colors属性设置成false来关闭。

```
colors: true,
```

11. hostname

主机名默认是localhost，如果想要改变它，可以设置hostname属性。

```
hostname: '127.0.0.1',
```

12. logLevel

当Karma里面什么东西出错了或者超出预期时，能看看更详细的信息会比较有用。我们可以设置logLevel属性来设置详细输出等级。当运行一个持续集成服务器时，很可能会要把日志输出整个关闭。

可能的日志值有：

- config.LOG_DISABLE
- config.LOG_ERROR
- config.LOG_WARN
- config.LOG_INFO
- config.LOG_DEBUG

```
logLevel: config.LOG_INFO,
```

13. 端口

Karma的Web服务器默认启动监听的端口是9876，可以在配置文件中自行修改这个端口。

```
port: 9875,
```

14. 预处理器

可以在测试运行之前让Karma预处理文件。当使用CoffeeScript[1]等语言来编写测试时，预处理就很有用了，它不再需要手工处理这些文件。

CoffeeScript预处理器默认集成在Karma中，但是其他预处理器就需要通过npm来请求额外的插件了。

Karma可用的预处理器包括：

- CoffeeScript

① http://coffeescript.org

- html2js

其他可以通过插件方式引入的预处理器有：

- coverage
- ng-html2js
- ember

想要包含它们中的一个或多个，可以通过npm命令来安装：

```
$ npm install karma-coverage --save-dev
```

要配置使用哪个预处理器，我们可以在配置文件映射里设置。默认的文件映射设置为{'**/*coffee': 'coffee'}。

```
preprocessors: {
    '**/*.coffee': ['coffee']
 }
```

也可以设置多个预处理器。配置依赖于我们使用的插件。比如我们要配置CoffeeScript：

```
coffeePreprocessor: {
  options: { bare: true }
}
```

我们也可以使用customPreprocessor属性来自定义预处理器。

```
customPreprocessor: {
  mini_coffee: {
    base: 'coffee',
    options: { bare: true }
  }
}
```

15. 代理

Karma可以设置HTTP代理，这样当我们的测试取到一个路由时，它们可以从远程服务器获取。这对于端到端测试（使用了服务器）来说是很有用的，也是需要的。

这个对象会是一个从路径指向远程服务器的键值对列表。

```
proxies: {
  '/': 'http://localhost:9000'
}
```

16. 报表

Karma的报表也是可以自定义的：可以把报表设置成在终端中显示关于所有类型测试状态的有用输出。

默认情况下，这个选项被设置成['progress']，它会以人类可读的形式报告测试过程。progress和dots都是Karma默认包含的报表。

也可以通过npm插件来包含其他报表，比如growl和coverage。我们使用npm来安装这些插件：

```
$ npm install karma-[plugin-name] --save-dev
```

17. `singleRun`

如果这个布尔值被设置成`true`，Karma会用所有已配置的浏览器运行这些测试一次；如果它们都通过了，就会看到一个退出代码0，要是有失败的，就会是1。

在持续集成服务器上运行我们的测试时，这个指标特别有用。

18. `urlRoot`

这个URL是Karma运行的根URL。我们可以用`urlRoot`参数来给Karma使用的一切URL加前缀。使用代理时，这是个好主意，这样来自我们测试的调用不会和服务器上已有的功能冲突。

19.8 使用 RequireJS

要在Karma中使用RequireJS，我们在karma.conf.js之后还需要一个额外的配置文件：test-main.js。

- ❑ karma.conf.js：负责Karma的配置（已经叙述）。
- ❑ test-main.js：负责给测试配置RequireJS。

1. karma.conf.js

我们会像平常一样用Karma配置文件生成器来配置`karma`：

```
$ karma init test/karma.conf.js
```

当提示使用RequireJS时，选择yes。

当生成器询问默认要加载哪个文件时，我们需要选择不用RquireJS加载的所有文件。只包含test/test-main.js也不会有问题，我们马上就创建它。

当我们列出源码和测试文件时，要选择所有要使用Require.js加载的文件。应当把用Require.js加载的所有文件都列出来，包括所有外部库、所有代码、所有测试文件。

我们需要用配置对象来配置这些文件，把它们设置成默认不包含。

这时，我们的karma.conf.js应该是：

```
module.exports = function(config) {
   config.set({
      basePath: '..',
      frameworks: ['jasmine', 'requirejs'],
      files: [
         {pattern: 'app/lib/angular.js', included: false},
         {pattern: 'app/lib/angular-route.js', included: false},
         {pattern: 'app/lib/angular-mocks.js', included: false},
         {pattern: 'app/js/**/*.js', included: false},
         {pattern: 'test/**/*.js', included: false}
         {pattern: 'test/lib/**/*.js', included: false},
         'test/test-main.js'
     ],
     exclude: [
       'js/main.js'
     ],
     reporters: ['progress'],
```

```
    port: 9876,
    colors: true,
    logLevel: config.LOG_INFO,
    autoWatch: true,
    browsers: ['Chrome'],
    captureTimeout: 60000,
    singleRun: false
 });
};
```

注意我们没有排除main.js，它是应用开始的文件。

鉴于Karma会提供app/js目录下的文件，我们来给Karma文件服务器配置一个起始的上下文，用相对路径加载模块。因为我们想要测试的baseUrl和源码文件在同一目录，所以需要把basePath设置成本地目录（.）。

2. test/test-main.js

我们的test-main.js文件会代替主应用文件，提供引用测试文件的能力，无需把应用真正启动起来。

Karma会包含所有在数组window.karma.files里面的文件，所以我们会在这里发现我们的测试文件。然后，就可以正常配置RequireJS了：

```
var tests = [];
for (var file in window.__ karma__.files) {
  if (window. __karma__.files.hasOwnProperty(file)) {
    if (/Spec\.js$/.test(file)) {
       tests.push(file);
  }}}

 requirejs.config({
   baseUrl: 'app',
   paths: {
     'jquery': 'lib/jquery',
     'angular': 'lib/angular',
     'angularRoute': 'lib/angular-route',
     'angularMocks': 'lib/angular-mocks',
   },
   shim: {
     'underscore': {
       exports: '_'
     }
  },

  // 让Require.js加载这些文件（我们所有的测试）
  deps: tests,
  // RequireJS完成之后就启动测试
  callback: window.__karma__.start
});
```

测试看起来将和默认不使用RequireJS的情况不太一样。可以简单地像平常一样使用RequireJS，把测试放在define()里。例如：

```
define([
  'app', 'jquery', 'angular',
  'angular', 'angularRoute', 'angularMocks'
```

```
    ],
    function() {
      describe('UnitTest: App', function() {
        // 跟平常一样
        it('is defined', function() {
          expect(_.size([1,2,3])).toEqual(3);
        });
      });
  });
```

19.9 Jasmine

我们来过一遍Jasmine测试框架。尽管Karma支持多种测试框架，但默认的选项是Jasmine。

Jasmine是一个用于测试JavaScript代码的行为驱动开发框架。既然我们将要用到Jasmine语法，先大致看一下怎样写基于Jasmine的测试套件吧。

19.9.1 细则套件

Jasmine套件的核心部分是describe函数。这个函数是Jasmine套件定义的一个全局函数，所以可以在测试中直接调用。

describe()函数带有两个参数，一个字符串，一个函数。字符串是待建立的细则（spec）套件名称或者描述，函数封装了测试套件。

```
describe('Unit test: MainController', function() {
});
```

可以嵌套这些describe()函数，这样我们可以创建一个测试树来执行那些在测试中设置的不同条件。

```
describe('Unit test: MainController', function() {
  describe('index method', function() {
    // 细则放这里
  });
});
```

使用describe()函数把相关的细则分组是个不错的主意。在每个describe()块运行时，这些字符串会沿着细则的名称链接起来。因此，上面这个例子的标题就会变成“Unit test: MainController index method.”

然后，这些describe()块的标题就会被追加到细则的标题上。设计这个步骤的目的是让我们以完整句子来阅读细则的，所以把测试命名成可读的英文就很重要了。

19.9.2 定义一个细则

我们通过调用it()函数来定义一个细则。这个函数也是在Jasmine测试套件中定义的全局函数，所以可以从测试中直接调用。

it()函数带有两个参数：一个字符串，是细则的标题或者描述；一个函数，包含了一个或多个用于测试代码功能的预期。

这些预期都是函数，执行时评估为true或false。一个所有预期都为true的测试就算是一条通过的细则，一条细则有一个或者多个预期为false的话，就是个失败的测试。

一个简单的测试可能像这样：

```
describe('A spec suite', function() {
   it('contains a passing spec', function() {
     expect(true).toBe(true);
   });
});
```

这个细则的标题，追加到describe()标题之后，就成为了“一个细则套件包含一条已通过的细则”。

19.10　预期

测试应用时，我们会想要断言条件在应用的不同阶段是符合我们期望的。我们要写的这个测试读起来就像这样：“如果我们点击这个按钮，就期望有这个结果。”例如，“如果我们导航到首页，我们期望欢迎信息会被渲染出来。”

使用expect()函数来建立预期。expect()函数带有一个单值参数。这个参数被称为真实值。

要建立一个预期，我们给它串联一个带单值参数的匹配器函数，这个参数就是期望值。

这些匹配器函数实现了一个在真实值和期望值之间的布尔比较。可以通过在调用匹配器之前调一个not来创建测试的否定式。

```
describe('A spec suite', function() {
  it('contains a passing spec', function() {
    expect(true).toBe(true);
   });
   it('contains another passing spec', function() {
     expect(false).not.toBe(true);
   });
});
```

Jasmine自带一大堆内置的匹配器，我们可以在测试应用的过程中使用。要写一个自定义的匹配器也很容易。

19.10.1　内置的匹配器

1. toBe

toBe()匹配器使用JavaScript操作符===来比较值：

```
describe('A spec suite', function() {
  it('contains passing specs', function() {
    var value = 10,
    another_value = value;
    expect(value).toBe(another_value);
    expect(value).not.toBe(null);
  });
});
```

2. toEqual

toEqual()匹配器比较的是值，对简单字面量和变量有效：

```
describe('A spec suite', function() {
  it('contains a passing spec', function() {
    var value = 10;
    expect(value).toEqual(10);
  });
});
```

3. toMatch

toMatch()匹配器使用正则表达式匹配字符串：

```
describe('A spec suite', function() {
    it('contains a passing spec', function() {
        var value = "<h2>Header element: welcome</h2>";
        expect(value).toMatch(/welcome/);
        expect(value).toMatch('welcome');
        expect(value).not.toMatch('goodbye');
    });
});
```

4. toBeDefined

toBeDefined()匹配器将值与undefined进行比较：

```
describe('A spec suite', function() {
   it('contains a passing spec', function() {
     var value = 10,
         undefined_value = undefined;
     expect(value).toBeDefined();
     expect(undefined_value).not.toBeDefined();
   });
});
```

5. toBeUndefined

toBeUndefined()匹配器的功能跟toBeDefined()匹配器相反：

```
 describe('A spec suite', function() {
   it('contains a passing spec', function() {
     var value = 10,
         undefined_value = undefined;
expect(undefined_value).toBeUndefined();
     expect(value).not.toBeUndefined();
   });
 });
```

6. toBeNull

toBeNull()匹配器将值与null进行比较：

```
describe('A spec suite', function() {
  it('contains a passing spec', function() {
    var value = null,
        not_null_value = 10;
    expect(value).toBeNull();
    expect(not_null_value).not.toBeNull();
```

```
    });
});
```

7. toBeTruthy

toBeTruthy()匹配器把值转换为布尔类型之后与true进行比较：

```
describe('A spec suite', function() {
  it('contains a passing spec', function() {
    var value = 10,
        undefined_value;
    expect(value).toBeTruthy();
    expect(undefined_value).not.toBeTruthy();
  });
});
```

8. toBeFalsy

toBeFalsy()匹配器把值转换成布尔类型之后与false比较：

```
describe('A spec suite', function() {
  it('contains a passing spec', function() {
    var value = 10,
        undefined_value;
    expect(undefined_value).toBeFalsy();
    expect(value).not.toBeFalsy();
  });
});
```

9. toContain

toContain()匹配器检测一个条目是否在数组中：

```
describe('A spec suite', function() {
  it('contains a passing spec', function() {
    var arr = [1,2,3,4];
    expect(arr).toContain(4);
    expect(arr).not.toContain(12);
  });
});
```

10. toBeLessThan

toBeLessThan()匹配器建立了一个期望，比较一个数值是否小于预期：

```
describe('A spec suite', function() {
  it('contains a passing spec', function() {
    var value = 10;
    expect(value).toBeLessThan(20);
    expect(value).not.toBeLessThan(5);
  });
});
```

11. toBeGreaterThan

toBeGreaterThan()匹配器建立了一个期望，比较一个数值是否大于预期：

```
describe('A spec suite', function() {
    it('contains a passing spec', function() {
       var value = 30;
       expect(value).toBeGreaterThan(40);
```

```
        expect(value).not.toBeGreaterThan(20);
    });
});
```

12. toBeCloseTo

toBeCloseTo()匹配器在一个指定的精度级别内比较一个值是否接近另一个值：

```
describe('A spec suite', function() {
  it('contains a passing spec', function() {
    var value = 30.02;
    expect(value).toBeCloseTo(30, 0);
    expect(value).not.toBeCloseTo(20, 2);
  });
});
```

13. toThrow

toThrow()匹配器验证一个函数是否抛出了异常：

```
 describe('A spec suite', function() {
   it('contains a passing spec', function() {
   expect(function() {
     return a + 10;
   }).toThrow();
   expect(function() {
     return 2 + 10;
   }).not.toThrow();
 });
});
```

14. 创建自定义匹配器

在代码中面对更复杂情况时，会需要创建自己的匹配器，Jasmine让这变得非常容易。要创建一个匹配器，我们可以在Jasmine块中调用addMatcher()函数，带入一个值：

```
describe('A spec suite', function() {
  this.addMatchers({
    toBeLessThanOrEqual: function(expected) {
      return this.actual <= expected;
    }
  });
});
```

然后就可以在测试套件里定义的任意测试中调用这个toBeLessThanOrEqual()匹配器了。

19.10.2 安装和卸载

除了手动在每个测试中设置测试条件，我们可以使用beforeEach方法来运行一组设置函数。beforeEach()函数带一个参数：一个函数，在每个细则运行之前被调用一次。它可以在一个描述块中使用，就像这样：

```
describe('A spec suite', function() {
  var message;
  beforeEach(function() {
    message = "hello ";
  });
  it('should say hello world', function() {
```

```
    expect(message + "world").toEqual("hello world");
  });
  it('should say hello ari', function() {
    expect(message + "ari").toEqual("hello ari");
  });
});
```

我们也可以重置条件（例如，使用`afterEach()`函数清除数据库，或者通过模拟冲掉所有请求）。与`beforeEach()`函数类似，它也带有一个参数：一个函数，会在每个细则跑完之后执行。

```
describe('A spec suite', function() {
  var count;
  afterEach(function() {
    count = 0;
  });
  it('should add one to count', function() {
    count += 1;
    expect(count).toEqual(1);
  });
  it('should check for the reset value', function() {
    expect(count).toEqual(0);
  });
});
```

在嵌套的描述块中，这些`beforeEach`和`afterEach`方法是被串起来的，所以我们可以建立更复杂的测试树，而无需重复代码。

19.11　端到端的介绍

做端到端测试时，我们会使用Angular场景运行器。Angular场景运行器模拟了用户交互，这样我们可以更准确地评估应用的状态。

编写场景测试时，我们要描述应用在不同情境下应有的行为。就像在单元测试里，我们也用Jasmine来建立期望和行为。

测试应用时，我们会直接使用场景运行器的API来控制浏览器。利用这个API，我们能通过不同的动作操作浏览器，包括在输入框中输入数据，选择元素，导航页面，控制浏览器的流，等等。

我们要使用的核心基础API是`browser()`方法，这个方法返回一个对象，为了控制浏览器，可以在这个对象上面串一些方法。

场景运行器通过打开一个浏览器窗口，嵌入一个iframe的方式来运行。这个iframe就是Karme运行应用测试，跟踪场景运行器成功或者失败结果的地方。

1. 导航页面

要在测试浏览器frame里面加载一个URL，我们使用`navigateTo`函数，它带有一个参数：要加载的URL。

```
browser().navigateTo(url)
```

我们也可以通过调用一个方法取得一个URL的方式来动态加载这个URL。这个调用一般用于我们在写测试或者检测某个操作的结果时，不知道目的URL的情况下。

```
browser().navigateTo(title, function() {
  // 在这里返回动态url;
   return '/';
});
```

2. 刷新页面

可以在测试frame里刷新当前加载的页面：

```
browser().reload()
```

3. 操作`window`对象

可以获取在测试frame里当前加载页面的超链接：

```
browser().window().href()
```

要获取测试frame中当前加载页面的路径，用下面这个语句：

```
browser().window().path()
```

要获取测试frame中当前加载页面的搜索字符串，执行：

```
browser().window().search()
```

可以像下面这样获取测试frame里当前加载页面最后一次的hash：

```
1 // 散列返回的时候不带#
2 browser().window().hash();
```

4. 位置、位置、位置

要获取测试frame中当前加载页面的`$location.url()`，我们用：

```
browser().location().url()
```

可以用这种方式获取测试frame中当前加载页面的`$location.path()`：

```
browser().location().path()
```

要像这样获取当前页面的`$location.search()`也是很容易的：

```
browser().location().search()
```

最后，也能获取到当前页面的hash：

```
browser().location().hash()
```

5. 建立预期

想真正校验我们的应用是按照期望来运行的，需要建立对某一状态的断言。我们可以用端到端和场景API的组合来做到这一点。

使用`expect()`，我们断言给定future对象是否与匹配器相符。场景API给出的任何返回值都是一个场景运行器将要解析的future对象，我们会校验这个最终的值是不是我们所期望的结果。

```
expect(browser().location().path())
  .toBe('/')
// 或者用not()来否定这个期望
expect(browser().location().path())
```

```
  .not().toBe('/home')
```

6. 跟内容交互

端到端测试特别强大，因为我们实际就在加载用户将要看到的页面，所以我们可以窥视他们所能看到的结果，并且验证它看上去是对的，并且以我们预期的样子在运行。

我们可以选择元素，在输入框中输入值，点击按钮，校验内容是否出现在该出现的地方，遍历循环器，等等。

要选择页面上的元素，使用element()方法。这个API带两个参数：

- 选择器—— jQuery HTML元素选择器；
- 标签——用于在浏览器或者终端输出的文本字符串。

```
element("form", "the signup form")
```

有了这个选中的元素，我们就可以执行方法来查询它在页面上的状态。要检测匹配给定jQuery选择器的元素数目：

```
element("input", "input elements").count()
```

要点击一个元素（比如一个提交按钮），可以调用：

```
element("button", "submit button").click()
```

可以使用query()方法在给定jQuery选择器上执行一个方法。

```
// 选择页面上所有的链接
element("a", "all links").query(
  // 所有这些链接
  // 都会作为元素传给函数
  function(elements, done) {
    // 对每个元素做些想做的事
    angular.forEach(elements, function(ele) {
      expect(ele.attr('ng-click'))
         .toBeDefined();
    });
    done(); // 告诉场景运行器我们做完了
  });
```

可以查阅每个元素，在jQuery属性上设置不同的期望。

可以获取或者设置一个元素的值：

```
element("button", "submit button").val()
// 设置
element("button", "submit button").val("Enter")
```

可以获取或者设置文本：

```
// 获取一块HTML的文本内容
element("h1", "header").text()
// 设置
element("h1", "header").text("Header text")
```

可以获取或者设置元素的HTML：

```
// 获取元素的HTML
```

```
element("h1", "header").html()
// 设置
 element("h1", "header").html("<h2>New header</h2>")
```

要设置或者获取高度：

```
// 获取元素的高度
element("div", "signup box").height()
// 设置
element("div", "signup box").height('200px')
```

要获取或者设置innerHeight：

```
// 获取元素的innerHeight
element("div", "signup box").innerHeight()
// 设置
element("div", "signup box").innerHeight('190px')
```

要设置或者获取outerHeight：

```
// 获取元素的outerHeight
element("div", "signup box").outerHeight()
// 设置
element("div", "signup box").outerHeight('210px')
```

要设置或者获取宽度：

```
// 获取元素的宽度
element("div", "signup box").width()
// 设置
element("div", "signup box").width('300px')
```

要设置或者获取innerWidth：

```
// 获取元素的'innerWidth'
element("div", "signup form").innerWidth()
// 设置
element("div", "signup form").innerWidth('200px')
```

要设置或者获取outerWidth：

```
// 获取元素的'outerWidth'
element("div", "signup form").outerWidth()
// 设置
element("div", "signup form").outerWidth('305px')
```

要设置或者获取元素的位置：

```
// 获取元素的位置
element(".logo", "our logo").position()
// 设置
element(".logo", "our logo").position("absolute")
```

要获取或者设置scrollLeft：

```
// 获取元素的scrollLeft
element("#signup_form", "signup form").scrollLeft()
// 设置
element("#signup_form", "signup form").scrollLeft(0)
```

要获取或者设置scrollTop的值，用这个值可以强制浏览器滚定到指定元素：

```
// 获取元素的scrollTop值
element("#signup_form", "signup form").scrollTop()
// 设置
element("#signup_form", "signup form").scrollTop(0)
```

要获取或者设置偏移量：

```
// 获取元素的偏移量
element("#signup_form", "signup form").offset()
// 设置
element("#signup_form", "signup form").offset(0);
```

也可以在jQuery选择器中查询或者变更一个元素的值，可以获取特性（使用`attr`）：

```
element("div", "signup box").attr('width')
// 设置
element("div", "signup box").attr('width', '100%')
```

可以获取一个属性（使用`prop`）：

```
element("div", "signup box").prop('width')
// 设置
element("div", "signup box").prop('width', '100%')
```

还可以获取CSS（使用`css`）：

```
element("div", "signup box").css('border-color')
// 设置
element("div", "signup box").css('border-color', 'red')
```

除了使用`element()`获取元素，还有其他与内容交互的方式。Angular的场景运行器包含了一些不同的帮助方法，能让我们查询和操作已渲染的DOM。

我们可以追溯自己所感兴趣的：Angular对不同元素的元素的认知。可以选中它们，找到绑定，与输入元素交互，查询页面以测试原生的Angular绑定。

7. 选择页面上的元素

场景运行器为我们建立的最底层帮助函数之一是`using()`函数。利用该函数，我们可以用jQuery类型的元素选择符定位指定的元素。

```
it('does not test anything yet', function() {
    // 定位指定元素
    using('.input_email').binding('email');
});
```

`using()`方法最多可带两个参数。

- jQuery选择器。我们用这个选择器来选定页面上的元素。
- 标签（可选的字符串）。这个字符串是一个标签，运行器用它在测试的输出中标识这个选择器。

8. 与Angular的绑定进行交互

场景运行器包含了一个途径，可以深入到Angular建立的绑定中，这样就可以从DOM上查询到Angular的绑定，然后从这个指定元素上选择第一个绑定关系。

例如，如果我们有一段HTML，在它上面包含有`$scope`元素的属性名：

```
<input type="text" ng-model="name" />
```

可以用binding()方法查询作用域中指定的绑定：

```
it('should update the name', function() {
  using('.form').input('name').enter('Ari');
  expect(
    using('.form').binding('name')
  ).toBe('Ari');
});
```

binding()方法带有一个名称参数，该参数是字符串类型。

这个字符串是我们在查询中所关注的DOM元素上的绑定名称。

9. 与输入元素交互

我们也可以跟页面上的输入元素交互。如果想要在一个文本框中输入文字，选中一个复选框，或者选择一个option元素的值，可以使用input()方法。

input()方法自身返回一个对象，我们可以调用这个对象的方法来跟元素进行交互。它带有一个名称参数，该参数是字符串类型。

这个名称是相应的ng-model的名称。

我们能从输入框上调用下列方法。

enter()。enter()方法向一个输入框输入值。

给定HTML：

```
<input type="text" ng-model="name" />
```

可以这样向输入框输入'Ari'：

```
input('name').enter('Ari');
```

check()。check()方法检测一个复选框的值。

给定HTML：

```
<input type="checkbox" ng-model="save" />
```

可以使用如下语句检测这个叫"save"的复选框：

```
input('name').check();
```

select()。select()方法选中一个单选按钮的指定值。

给定HTML：

```
<input type="radio" ng-model="color" value="red" />
<input type="radio" ng-model="color" value="blue" />
<input type="radio" ng-model="color" value="yellow" />
```

可以用这样的测试来选择单选按钮：

```
input('color').select('red');
```

val()。最后，可以简单地通过调用输入元素的.val()来获取输入框的当前值。我们会用这

个来检验指定输入元素的当前值。

```
input('color').select('red');
input('color').val(); //颜色将是"red"
```

19.11.1　选项输入

要从给定的选项输入框上选中指定的option值也很容易。我们会使用select()方法从select标签上选择一个option。

给定HTML：

```
<select ng-model="color"
      ng-options="c.name for c in colors">
    <option value="">Pick your favorite color"</option>
</select>
```

然后是JavaScript：

```
select('color')
```

select()方法返回一个对象，带有一个方法，可以用于选择这个select元素的一个选项。它也让我们能在多选select中选取多个项。

option()。option()方法能让我们选中列表中的一个值。

```
select('color').option('red');
```

option()方法带有一个值参数，该参数是字符串类型。

这个value参数是一个字符串，可以让select选中给定的值。

options()。options()方法能让我们选中多选select中的多个值。

```
select('color').options('Ghostbusters', 'Titanic');
```

在必要的情况下，为了选中option的值，options()方法可以带任意数量的参数，这时参数是一组字符串。

这组字符串是要从多选select中选择的值。

19.11.2　重复循环元素

Angular通过ng-repeat指令，使从列表创建DOM元素变得非常容易，Angular场景也让我们能更容易测试这些循环指令。

repeater()函数自身返回一个对象，带有多个方法，用这些方法可以查询视图中的一组元素。它最多可带两个参数。

- 选择器（字符串）。jQuery选择器，指向那些我们所关注的元素。
- 标签（字符串，可选）。标签是用于测试输出的一个字符串。

下面列出了可对重复器返回的一组元素进行调用的方法。对于每个测试，我们用下面这段HTML作示例：

```
<table id="phonebook">
  <tr ng-repeat="person in people">
    <td>{{ person.name }}</td>
    <td>{{ person.email }}</td>
   </tr>
</table>
```

方法如下。

count()。count()方法返回重复器里有多少行与DOM中的jQuery选择器匹配。

```
repeater('#phonebook tr').count();
```

count方法不带参数，就简单地返回一个整数。

column()。column()方法返回一个数组，数组中的元素是列中的值，这些值包含了与DOM中jQuery选择器匹配的重复器中的给定绑定。

```
repeater('#phonebook tr')
    .column('person.name');
```

column()方法带有一个字符串类型的绑定参数， 这个绑定是针对重复器中指定元素的。它是在元素中渲染的绑定的名称。

row()。row()方法返回一个数组，数组中的元素是行中的值，这些值包含了与DOM中给定jQuery选择器匹配的重复器中的给定绑定。

```
repeater("#phonebook tr").row(0);
```

row()方法带有一个整形的索引参数。

index是要从中返回给定绑定的列的序号。

19.12 模拟和测试帮助函数

开始写测试之前，我们需要理解测试的一个核心特性：模拟。在测试中，模拟是一个古老的概念，允许我们在受控环境下定义模拟对象来模仿真实对象的行为。

AngularJS提供了它自己的模拟库，称为angular-mocks，它位于angular-mock.js文件中。模拟对象是专门设计用于单元测试的。

要在单元测试中建立模拟对象，需要确保在Karma配置中包含了angular-mock.js文件。

我们必须确保test/karma.conf.js文件的files数组中包含了angular-mock.js。包含了这个依赖之后，就可以创建Angular模块的模拟引用了。

例如，在一般的单元测试设置里，我们会创建一个describe执行环境，在每个测试在describe的上下文中运行之前，我们在这个执行环境中调用angular.mock.module：

```
describe('myApp', function() {
  // 模拟'myApp' angular模块
  beforeEach(angular.mock.module('myApp'));

  it('...')
});
```

> 注意，我们只要调用module就可以了，因为angular.mock.module函数被发布在全局作用域的window接口上了。

建立了模拟的Angular模块之后，可以把连接到这个模块上的任意服务注入到我们测试代码里。

凭借这些测试，我们需要像Angular那样在运行时注入依赖关系。在我们的单元测试中，这一步是必要的，因为我们隔离了想要测试的功能。

要注入一个依赖，在beforeEach函数调用中使用angular.mock.inject方法，类似之前做的那样。

```
describe('myApp', function() {
  var scope;

  // 模拟我们的'myApp' angular模块
  beforeEach(angular.mock.module('myApp'));
  beforeEach(angular.mock.inject(function($rootScope) {
    scope = $rootScope.$new();
  });
  it('...')
});
```

> 类似于module函数，inject函数也是在window对象上的，为的是全局访问，所以也可以直接调用inject。

在这个测试中，就像在其他几乎所有单元测试中那样，我们想要保存当前工作对象实例的引用（在上面例子中，保存的是scope）。那样，我们可以在整个it()子句中对这个对象引用进行操作。

通常，我们会用将引用注入进测试时使用的名字来保存它。比如，如果我们在测试一个服务，可以注入这个服务，然后把它的引用用一种稍微不同的命名方案存储起来。我们想在注入的服务名称两端使用下划线，这样当它被注入时， 注入器会忽略它的名称。

```
describe('myApp', function() {
  var myService;

  // 模拟我们的'myApp'angular模块
  beforeEach(module('myApp'));
  beforeEach(inject(function(_myService_) {
    myService = _myService_;
  });
  it('...')
});
```

19.13　模拟$httpBackend

Angular也内置了$httpBackend模拟库，这样我们可以在应用中模拟任何外部的XHR请求，避免在测试中创建昂贵的$http请求。

$httpBackend服务是一个假的HTTP后端实现，能让我们隔离和指定外部服务器可能处于的条件，这样我们可以精确确定应用在不同条件下的行为。

使用$httpBackend，我们可以校验一个请求的产生，对响应打桩、基于远程服务器的响应

来设置断言，用于校验对应用行为的期望。$httpBackend仅在单元测试中使用。

> 在端到端测试中也可以用$httpBackend服务，但是这么做一般测不全整个应用，因为没有使用真正的服务器。

使用$httpBackend的测试可行是因为劫持了依赖注入链：我们注入了模拟的$httpBackend，而不是使用$http服务产生实际HTTP请求的正版$httpBackend服务。这样就不需要为了支持测试而修改应用。

1. 冲刷HTTP请求

在生产中，$httpBackend异步响应请求，这在测试环境中基本很难配置。因而，我们需要在测试的最后手动冲刷一切挂起的请求，这样才能清理仍然保持了$httpBackend异步行为的执行环境。

$httpBackend带有两个方法，用于配置模拟的后端系统来处理HTTP响应，这两个方法是except和when，它们有不同的使用场景。

通常，在一个单元测试中，我们要确保配置的所有请求最终都按照预期运行了，如果没有的话就抛出异常。此外，还要确保每个测试结束时，不会仍有未结束的请求挂起。

可以在一个afterEach块中用两个方法来处理这两种情况：

```
// ...
afterEach(function() {
  $httpBackend.verifyNoOutstandingExpectation();
  $httpBackend.verifyNoOutstandingRequest();
});
```

有的情况下，我们要重置所有已设置请求的预期。要在一个多阶段测试内部复用$httpBackend的同一实例时，会出现这种情况。

可以用resetException()方法来重置它们：

```
// ...
it('should be a multiple-phase test', function() {
  // ...
  $httpBackend.resetExpectations();
  // ...
});
```

2. expect

except方法建立了一个请求的期望，用于对应用产生的请求作出断言，也用于定义它们的响应。如果预期的请求没有产生，或者不正确地产生了，测试就失败了。这些请求预期用于建立断言：请求已被产生。

except方法带有两个必选参数、两个可选参数。

- method：字符串HTTP方法，就像"GET"或者"POST"。
- url：期望调用的HTTP URL或者是一个函数接受给定URL并返回一个标识它是否匹配的布尔值。如果匹配它应该返回true，否则返回false。

- data（可选）：HTTP请求的主体，或者是个函数，接受一个data字符串并且在data符合预期时返回true（或者是一个用JSON格式发送HTTP主体的JavaScript对象）；
- headers（可选）：HTTP头或者函数，该函数接收header对象作为参数，并且在headers匹配预期时返回true。

except方法返回一个对象，该对象的respond方法用于控制在测试中如何处理匹配请求。

```
describe('Remote tests', function() {
  var $httpBackend, $rootScope, myService;

  beforeEach(inject(
      function(
        _$httpBackend_, _$rootScope_, _myService_) {
     $httpBackend = _$httpBackend_;
     $rootScope = _$rootScope_;
     // myService是一个服务
     // 为我们产生HTTP调用
     myService = _myService_;
  }));

  it('should make a request to the backend', function() {
     // 建立一个预期
     // myService会向路由发送一个GET请求
     // /v1/api/current_user
     $httpBackend.expect('GET', '/v1/api/current_user')
       .respond(200, {userId: 123});
     myService.getCurrentUser();
     // 冲刷请求很重要
     $httpBackend.flush();
   });
 });
```

$httpBackend.expect方法带有几个帮助函数，让我们能更加具体地描述设置的预期。

expectGET()为GET方法创建了一个新的请求预期，expectGET()带有两个参数。

- url：一个HTTP URL或者接受URL的函数，并且该URL匹配当前定义时返回true。
- headers（可选）：HTTP头。

```
// ...
$httpBackend.expectGET("/v1/api/current_user")
```

expectHEAD()为HEAD方法创建了一个新的请求预期。它带有两个参数。

- url：一个HTTP URL或者接受URL的函数，并且该URL匹配当前定义时返回true。
- headers（可选）：HTTP头。

```
// ...
$httpBackend.expectHEAD("/v1/api/current_user")
```

expectJSONP()为JSONP请求创建了一个新的请求预期。它只带有一个参数。

- url：一个HTTP URL或者接受URL的函数，并且该URL匹配当前定义时返回true。

```
// ...
$httpBackend.expectJSONP("/v1/api/current_user")
```

expectPATCH()为PATCH请求创建了一个新的请求预期。它接受三个参数。

- url：一个HTTP URL或者接受URL的函数，并且该URL匹配当前定义时返回true。
- data（可选）：HTTP请求的主体，或者是个函数，接受一个data字符串并且在data符合预期时返回true，或者是一个用JSON格式发送HTTP主体的JavaScript对象。
- headers（可选）：HTTP头。

```
// ...
$httpBackend.expectPATCH("/v1/api/current_user")
```

expectPOST()为POST请求创建了一个新的请求预期。它带有三个参数。

- url：一个HTTP URL或者接受URL的函数，并且该URL匹配当前定义时返回true。
- data（可选）：HTTP请求的主体，或者是个函数，接受一个data字符串并且在data符合预期时返回true，或者是一个用JSON格式发送HTTP主体的JavaScript对象。
- headers（可选）：HTTP头。

```
// ...
$httpBackend.expectPOST("/v1/api/sign_up", {'userId': 1234});
```

expectPUT()为PUT请求创建了一个新的请求预期。它带有三个参数。

- url：一个HTTP URL或者接受URL的函数，并且该URL匹配当前定义时返回true。
- data（可选）：HTTP请求的主体，或者是个函数，接受一个data字符串并且在data符合预期时返回true，或者是一个用JSON格式发送HTTP主体的JavaScript对象。
- headers：（可选）HTTP头。

```
// ...
$httpBackend.expectPUT("/v1/api/user/1234", {'name': 'Ari'});
```

expectDELETE()为DELETE请求创建了一个新的请求预期。它带有两个参数。

- url：一个HTTP URL或者接受URL的函数，并且该URL匹配当前定义时返回true。
- headers：（可选）HTTP头。

```
// ...
$httpBackend.expectDELETE("/v1/api/user/123")
```

3. requestHandler

我们的expect()方法都会返回一个requestHandler对象，带有一个函数：respond。respond方法让我们能给模拟的HTTP请求建立一个响应。

requestHandler的response函数有两种形式。

第一种形式允许我们设置响应代码、响应数据、响应头，或者全部三项。

```
// ...
$httpBackend.expectGET("/v1/api/current_user")
  // 响应一个200状态代码
  // 还有主体“success”
  .respond(200, 'Success');
  // 或者只返回数据
  .respond("Fail");
  // 或者只有请求头
  .respond({'X-RESPONSE', 'Failure'});
```

第二种形式能让我们设置一个请求处理程序，请求成功执行之后会执行它。这种形式不返回数据，它返回的是一个函数，能返回一个包含响应状态代码、响应数据和响应头的数组。

```
// ...
$httpBackend.expectGET("/v1/api/current_user")
  // 响应一个200状态代码
  // 还有主体“success”
  .respond(function(method, url, data, headers) {
    return [200, "DATA", {"header1": "Header1"}];
  });
```

4. when

$httpBackend也有when方法，与expect方法不同，它压根就没有对请求创建预期。实际上，它的目的主要是给应用创建一个假的后端，返回假数据。

不同于预期，使用when()时，每个匹配URL的请求都会被一条when定义处理。此外，用expect时，响应不是必须的，但用when时响应必须有。

如果要建立对所有测试通用的后端定义，那么使用when()方法是非常棒的。（例如，当测试一个使用了resolve属性的控制器时，它会依赖于外部数据的加载。）

when()函数带有两个必选参数和两个可选参数。

- method：字符串HTTP方法，就像"GET"或者"POST"。
- url：期望调用的HTTP URL。
- data（可选）：HTTP请求的主体，或者是个函数，接受一个data字符串并且在data符合预期时返回true，或者是一个用JSON格式发送HTTP主体的JavaScript对象。
- headers（可选）：HTTP头或者函数，会接受header对象，并且在headers匹配预期时返回true。

```
// ...
$httpBackend.when('GET', "/v1/api/current_user")
  // 响应一个200状态代码
  // 还有主体“success”
  .respond(200, 'success');
```

类似于expect方法，我们也有同样的帮助方法让when的使用更具描述性。

whenGET()为GET方法创建了一个新的后端定义，whenGET()带有两个参数。

- url：一个HTTP URL。
- headers（可选）：HTTP头。

```
// ...
$httpBackend.whenGET("/v1/api/current_user")
  .respond(200, {userId: 123});
```

whenHEAD()为HEAD方法创建了一个新的后端定义，whenHEAD()带有两个参数。

- url：一个HTTP URL。
- headers（可选）：HTTP头。

```
// ...
$httpBackend.whenHEAD("/v1/api/current_user")
  .respond(200);
```

whenJSONP()为JSONP请求创建了一个新的后端定义。它只带有一个参数。

- url：一个HTTP URL。

```
// ...
$httpBackend.whenJSONP("/v1/api/current_user")
  .respond({userId: 123});
```

whenPOST()为POST请求创建了一个新的后端定义。它带有三个参数。

- url ：一个HTTP URL。
- data（可选）：HTTP请求的主体，或者是个函数，接受一个data字符串并且在data符合预期时返回true，或者是一个用JSON格式发送HTTP主体的JavaScript对象。
- headers：（可选） HTTP头。

```
// ...
$httpBackend.whenPOST("/v1/api/sign_up",
  {'userId': 1234})
  .respond(200);
```

whenPUT()为PUT请求创建了一个新的后端定义。它带有三个参数。

- url：一个HTTP URL。
- data （可选：HTTP请求的主体，或者是个函数，接受一个data字符串并且在data符合预期时返回true，或者是一个用JSON格式发送HTTP主体的JavaScript对象。
- headers （可选）：HTTP头。

```
// ...
$httpBackend.whenPUT("/v1/api/user/1234", {'name': 'Ari'});
```

whenDELETE()为DELETE请求创建了一个新的后端定义。它带有两个参数。

- url：一个HTTP URL。
- headers -（可选） HTTP头。

```
// ...
$httpBackend.whenDELETE("/v1/api/user/123")
  .respond(200);
```

19.14 测试一个应用

测试机制建立起来之后，就可以开始测试应用的不同组件了。模块中只要包含了有可能发生变化的逻辑，那这些部分都很适合测试。路由是按照我们期望的那样运作的吗？页面包含指定内容吗？控制器代码执行了吗？

我们要关注：测试应用的不同组件，最常见的测试应用的地方，还有测试不同组件的经验技巧。

我们会测试应用中的下列组件：

- 路由；
- 请求和页面内容；

- ❑ 控制器；
- ❑ 服务与工厂；
- ❑ 过滤器；
- ❑ 模板和视图；
- ❑ 指令；
- ❑ 资源；
- ❑ 动画。

对于每个组件，我们会看看测试的可选项，然后探讨如何用可行的方法来测试它们。

对于多数测试，基准代码看起来像这样：

```
describe('NAME', function() {
});
```

19.14.1　测试路由

测试路由时，要建立一个测试来确保应用正确地把请求导到我们感兴趣的路由去了。我们需要检测路由是否在运作，是否找到了，或者是404了。我们要确认路由事件触发了，预期的模板是否真的加载了。

我们可以使用单元测试或者端到端测试来测试路由。既然路由会改变页面的地址（URL）和页面内容，我们需要检测路由是否被加载了，页面是否找到了，在这中间发生了什么。

要测试这些路由，我们假定建立了这么下面一个简单的路由代码：

```
angular.module('myApp', ['ngRoute'])
  .config(function($routeProvider) {
     $routeProvider
        .when('/', {
          templateUrl: 'views/main.html',
          controller: 'HomeController'})
        .when('/login', {
          templateUrl: 'views/login.html',
          controller: 'LoginController'})
       .otherwise({redirectTo '/'});
})
```

1. 单元测试路由

为了建立用于测试路由代码的单元测试，我们需要做下面几件事。

- ❑ 注入$route、$location和$rootScope服务。
- ❑ 建立一个模拟的后端来处理XHR，获取模板代码。
- ❑ 设置一个地址，运行一个$digest生命周期。

我们要存储这三个服务的一个副本用于测试（location、route和rootScope），这样以后就可以在测试中引用这些服务。

```
describe('Routes test', function() {
  // 在测试中模拟我们的模块
  beforeEach(module('myApp'));
```

```
    var location, route, rootScope;
    beforeEach(
      inject(_$location_, _$route_, _$rootScope_) {
        location = _$location_;
        route = _$route_;
        rootScope = _$rootScope_;
     });
     // 我们的测试代码放在这里
   });
```

既然已经把服务注入到控制器里了，我们来设置一个模拟后端，从templateUrl获取模板。可以用$httpBackend来创建断言来判断指定模板被加载了：

```
describe('Routes test', function() {
  // 在测试中模拟模块
  beforeEach(module('myApp'));

  var location, route, rootScope;
  beforeEach(inject(
    function(_$location_, _$route_, _$rootScope_) {
      location = _$location_;
      route = _$route_;
      rootScope = _$rootScope_;
    }));
  describe('index route', function() {
  beforeEach(inject(
      function($httpBackend) {
         $httpBackend.expectGET('views/home.html')
            .respond(200, 'main HTML');
    }));
     // 我们的测试代码放在这里
 });
});
```

测试代码都建立完了，现在可以开始写测试了！

为了用单元测试来测试路由，我们需要模拟路由在生产中的运作。路由借助于digest生命周期来运行，当location被设置以后，它使用一个digest循环周期来处理路由，改变页面内容，然后结束本次路由。了解了这些之后，我们就需要解释测试中路径的变更。

在测试中，我们打算在应用的index路由上测试两个状态。

- 当用户导航到首页时，指定的控制器会给他们显示首页。
- 当用户导航到一个未知路由时，他们会按照在otherwise函数中定义的那样被带到首页。

我们可以通过建立$location服务来传递路径的方式测试这些条件。为了触发location请求，我们要运行一个digest周期（在$rootScope上），然后检测控制器是符合预期的（在本例中，是“HomeController”）。

```
it('should load the index page on successful load of /',
function() {
  location.path('/');
  rootScope.$digest(); // 调用digest循环
  expect(route.current.controller)
    .toBe('HomeController')
  });
  it('should redirect to the index path on non-existent
     route', function() {
```

```
        location.path('/definitely/not/a/_route');
        rootScope.$digest();
        expect(route.current.controller)
          .toBe('HomeController')
});
```

要运行这些测试，需要确保Grunt服务器在运行：

```
$ cd myApp
$ grunt server
```

如果在应用文件中运行karma start karma.conf.js，能立刻在终端看到输出，如图19-3所示。

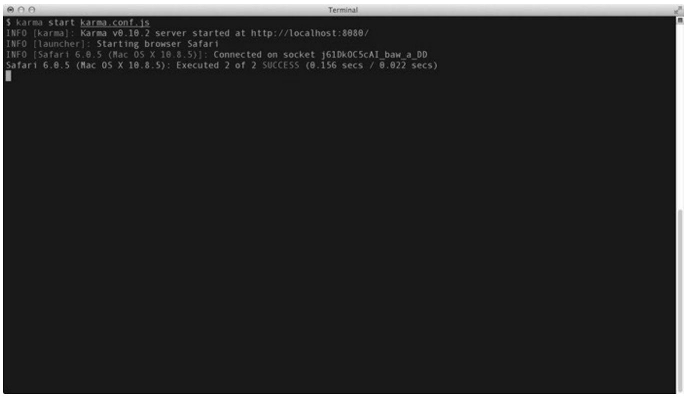


图19-3　单元测试路由

我们做了很多工作来建立路由测试，却只测试了一个路由地址。既然我们是为用户功能测试流程的变化，可以把这些工作移到应用上，在端到端测试中更严格地测试它。

2. 端到端的路由测试

在端到端测试中，无须模拟Angular的任何部分：我们是在黑盒测试这个应用。用这种方式，只需描述我们想要应用如何表现，并相应编写测试。

在编写端到端测试时，需要思考用户是如何在我们的应用中进行导航的。我们的测试应当是可读的：把用户带到特定页面，描述他们在应用中应当有什么样的体验。

所有端到端测试的基准测试只有下面几行：

```
describe('E2E: NAME', function() {
  // 我们的测试代码放在这里
});
```

就是这样。我们来使用browser() API方法在浏览器中修改iframe的源。

要测试index路由，我们要把浏览器指向index路由，然后确认地址也真的就在首页。

```
describe('E2E: Routes', function() {
  it('should load the index page', function() {
     browser().navigateTo('/#/');
     expect(browser().location().path()).toBe('/');
   });
 });
```

要运行这个测试，需要确保Grunt服务器在运行：

```
$ cd myApp
$ grunt server
```

然后运行Karma：

```
$ karma start karma-e2e.conf.js
```

这里，我们会立即在终端中看到输出。如果测试成功了，能看到它通过了所有测试，如果没有，会报告失败，如图19-4。

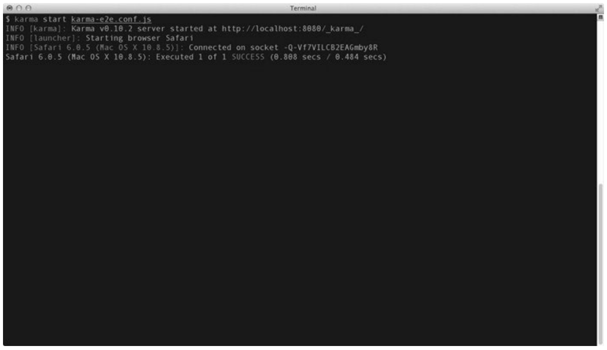


图19-4 在终端中端到端测试路由

我们也可以用浏览器来调试端到端测试。启动karma时，在后台开了一个浏览器。打开这个浏览器，点击右上角的debug按钮。点击这个按钮会打开一个新页面，显示我们所有的测试，包一组通过的测试和一组没有通过的测试。开发测试时，可以把使用浏览器作为调试应用和测试的重要参考，如图19-5所示。



图19-5 测试在浏览器中的可视化呈现

19.14.2 测试页面内容

测试时，需要确保页面的内容被浏览器正确渲染了。我们需要断言某内容被发送给浏览器了，并且最终展示给用户了。

通过浏览器中的单元测试，我们无法深入了解应用状态，因为在单元测试中不直接访问浏览器的内容。

我们可以确认控制器在执行预期的功能，设置断言来确认内容被加载，后面会深入探讨这些。

端到端测试页面内容

从另一方面讲，端到端测试对于为已加载的HTML建立预期来说是很理想的。利用端到端测试，应用里每个负责执行成功浏览器请求的部分都会被触发。

要建立端到端的内容测试，我们跟往常一样建立测试：

```
describe('E2E: Content', function() {
});
```

在这个路由测试中，我们将建立两个场景：

- 在首页上，有一个指向登录路由的链接；
- 可以点击这个链接，然后它会带我们到登录页面。

在第一个测试中，我们只验证页面上存在一个登录按钮。我们来建立一个断言：有一个登录按钮可以单击。

在index.html里，假定有如下内容：

```
<div id="authorize">
  <a id="login" class="radius" href="#/login">Try it! Sign in</a>
</div>
```

在第一个测试中，我们只确认存在匹配按钮上文字的元素：

```
it('should have a sign up button', function() {
  browser().navigateTo('/#/');
  expect(
    element("a#login").html()
  ).toEqual("Try it! Sign in");
});
```

现在，要保证点击这个链接能把用户带到登录页。我们建立另外一个测试来断言如果我们点了这个链接，新的地址就是login路由：

```
it('should show login when clicking sign in', function() {
  browser().navigateTo('/#/');
  element("a#login", "Sign in button").click();
  expect(browser().location().path())
    .toBe('/login');
});
```

最后，如果我们要针对一个用户作测试的话，可以建立测试，通过选出输入元素，设置值的方式来填充登录表单，如图19-6所示。

```
it('should be able to fill in the user info',
  function() {
    browser().navigateTo('/#/');
    element("a#login", "Sign in button").click();
    input("user.email").enter("ari@fullstack.io");
    input("user.password").enter('123123');
    element('form input[type="submit"]').click();
    expect(browser().location().path())
      .toBe('/dashboard');
});
```

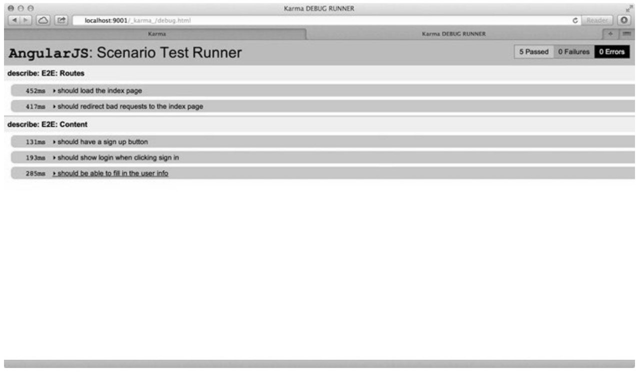


图19-6　内容加载测试

19.14.3　测试控制器

控制器包含了应用的业务逻辑。控制器是$scope把控制与视图关联起来的地方。既然我们要在控制器中做应用内的多数更新视图的工作,那就要建立测试来保证它们的行为是按预期执行的。

1. 单元测试控制器

当对控制器作单元测试时,需要建立测试来模拟Angular的行为。

在建立单元测试的过程中,需要确保:

- 建立了测试来模拟模块;
- 用一个已知的作用域实例来存储控制器的一个实例;
- 基于作用域来测试我们的预期。

要初始化一个控制器实例,需要使用$new()方法从$rootScope创建某作用域的一个新实例。这个新实例会建立Angular在运行时使用的作用域继承。

有了这个作用域,就可以初始化一个新的控制器,把这个作用域作为控制器的$scope传递过去。

```
describe('Unit controllers: ', function(){
  // 模拟myApp模块
  beforeEach(module('myApp'));
  describe('FrameController', function() {
    // 局部变量
    var FrameController, scope;
    beforeEach(inject(
      function($controller, $rootScope) {
        // 创建子作用域
        scope = $rootScope.$new();
        // 创建FrameController的新实例
        FrameController = $controller('FrameController',
          { $scope: scope });
     }));

   // 我们的测试代码放在这里
   });
 });
```

测试建立好了之后，我们既有了FrameController的一个实例，也有了这个控制器的$scope。现在可以用这个scope来测试FrameController上的作用域了。

在FrameController里，我们有一个时钟在应用顶部显示当前时间。我们也可以访问一个用户和他的时区。

控制器代码的相关部分看起来像这样：

```
angular.module('myApp.controllers', [])
  .controller('FrameController',
      function($scope, $timeout) {
           $scope.time = {
              today: new Date()
         };
         $scope.user = {
              timezone: 'US/Pacific'
         }
          var updateClock = function() {
              $scope.time.today = new Date();
          };
         var tick = function() {
              $timeout(function() {
                  $scope.$apply(updateClock);
                  tick();
             }, 1000);
          }
           tick();
  });
```

为了测试关注点，我们的FrameController已经刻意简化过了。要看到示例应用的完整测试套件，可以查阅本书附带的代码。

我们会测试控制器的两个功能：

- 时间已被定义；
- 用户已被定义，并且有时区。

```
// 测试FrameController的值
it('should have today set', function() {
   expect(scope.time.today).toBeDefined();
});

it('should have a user set', function() {
    expect(scope.user).toBeDefined();
});
```

2. 端到端测试控制器

对控制器的端到端测试结果看上去与测试页面内容按预期渲染的结果很像：我们要测试控制器里的所有函数确实触发了。

要测试页面是不是被渲染了，可以在浏览器中加载相关页面，测试我们所期待的内容是否在视图中渲染了。

端到端测试的样板看起来如下所示：

```
describe('E2E controllers: ', function() {
  // 我们的测试代码放在这里
});
```

测试建立好了之后，就可以添加细则了。我们要测试日期（或者至少它的一部分）存在于页面上，时区也在。

```
beforeEach(function() {
  browser().navigateTo('/#/');
});

it('should have the date in the browser', function() {
  var d = new Date();
  expect(
    element("#time h1").html()
  ).toMatch(d.getFullYear());
});

it('should have the user timezone in the header', function() {
  expect(
    element('header').html()
  ).toMatch('US/Pacific');
});
```

想知道timeout函数在控制器上被调用了，也比较方便。可以使用Jasmine帮助函数createSpy来确认timeout确实被调用了。

如果修改了beforeEach()函数（看上面），可以在控制器中包含$timeout服务。

```
var FrameController, scope, timeout;
beforeEach(inject(
  function($controller, $rootScope) {
    scope = $rootScope.$new();
    timeout = jasmine.createSpy('timeout');
    FrameController = $controller('FrameController', {
      $scope: scope,
      $timeout: timeout
    });
}));
```

现在，在测试中可以设置一个预期：服务确实被调用了：

```
it('should set the clock a foot', function() {
    expect(timeout).toHaveBeenCalled();
});
```

19.14.4 测试服务和工厂

服务是很容易测试的：它们是独立的对象，提供了本地化的功能。既然它们是单例对象，我们可以隔离地创建这些对象，测试它们的行为。

1. 单元测试服务

单元测试服务非常容易，我们只要把服务注入到测试中就可以了。

用一个简单例子开始吧，设想我们有一个服务提供了版本信息：

```
angular.module('myApp.services', [])
  .value('version', '0.0.1');
```

在这个情况下，服务提供了一个字符串值。我们可以把这个版本服务注入到当前测试中。

单元测试的样板看上去像这样：

```
describe('Unit: services', function() {
  beforeEach(module('myApp'));
});
```

> 到现在为止，我们已经隐式调用过$injector服务了。本例展示了如何显式调用它。

为隔离这个测试，我们把它嵌套在一个describe()块中，并且在beforeEach()块中注入了版本服务。

```
describe('version', function() {
  var version;
  beforeEach(inject(function($injector) {
    // 使用$injector获取版本服务
    version = $injector.get('version');
  }));

  it('should have the version as a service',
    function() {
      // 设置对版本服务的预期
      expect(version).toEqual('0.0.1');
  });
});
```

正如我们所看到的，测试服务真是简单，然而，我们的服务并不总是这么简单。在示例应用中，我们跟googleApi交互，这个服务就有些复杂了。

这是googleService.googleApi服务的完整源码：

```
// google服务模块
angular.module('googleServices', [])
  .factory('googleApi',
    function($window, $document, $q, $rootScope) {
      // 创建了一个defer
      // 封装了我们的Google API服务的加载
      var d = $q.defer();

      // 脚本被浏览器加载之后，
      // 我们就要调用这个函数了，
      // 它反过来会解析我们的全局defer
      $window.bootGoogleApi = function(keys) {
        // 需要设置我们的API key
        window.gapi.client.setApiKey(keys.apiKey);
        $rootScope.$apply(function() {
          d.resolve(keys);
        });
      };

      // 在浏览器中加载客户端
      var scriptTag = $document[0].createElement('script');
      scriptTag.type = 'text/javascript';
```

```
    scriptTag.async = true;
    scriptTag.src =
      'https://apis.google.com/js/client:plusone.js?onload=onLoadCallback';
    var s = $document[0].getElementsByTagName('body')[0];
    s.appendChild(scriptTag);

    // 返回一个单例对象
    // 这个对象返回一个promise
    return {
      gapi: function() { return d.promise; }
    }
});
```

服务自身返回了一个对象，包含一个返回promise的函数。这个promise会在Google API被加载到页面上并准备好之后被解析一次。

要为这个服务建立预期，需要使用Jasmine的spyOn方法来在我们调用的方法上创建一个间谍，并且建立一个期望：它确实被调用了。

我们会在一个隔离的describe()块中建立对Google API的测试：

```
describe('googleServices', function() {
  var googleApi, resolvedValue;

  beforeEach(inject(function($injector) {
    // 从我们的服务中获取定义的googleApi
    googleApi = $injector.get('googleApi');
    // 为gapi函数创建一个间谍
    // 用于告诉我们它何时被调用
    // 但不阻止真正的函数调用
    spyOn(googleApi, 'gapi')
      .andCallThrough();
    // 使用真正函数的resolve
    // 来设置解析之后的值
    googleApi.gapi().then(function(keys) {
      resolvedValue = keys;
    });
  }));

  describe('googleApi', function() {
      // 我们的测试放在这里
  });
});
```

我们也可以使用名称来注入上面的googleApi服务，因为inject()函数使用与$injector相同的语法。上面的调用会变成inject(function(googleApi))。

> 使用andCallThrough()方法，我们的测试就会等到gapi对象在window上就绪。可以用一个不同的方法andCallFake()来给这些请求打桩。

```
var q;
 beforeEach(inject(function($injector) {
   // 从服务中获取预定义的googleApi
   googleApi = $injector.get('googleApi');
   // 取得$q对象
   q = $injector.get('$q');
   // 为gapi函数创建一个间谍
```

```
    // 并且模拟真实的响应
    spyOn(googleApi, 'gapi')
      .andCallFake(function() {
        var d = q.defer(); // 模拟deferred函数
        setTimeout(function() {
          resolvedValue = {
            clientId: '12345'
          }
        }, 100);
        return d.promise;
      });
      // 使用真正函数的resolve
      // 来设置解析之后的值
      googleApi.gapi().then(function(keys) {
        resolvedValue = keys;
      });
  }));
```

现在，可以用我们的间谍来判断函数是否确实被调用了。

在这个`describe()`块中的第一个测试只是简单地测试了这个方法存在，并且是一个函数。如果我们还在用这个API，并且改变了方法名或者签名，这个测试就会很有用。

我们用Jasmine帮助函数`waitsFor()`建立测试来等待一个promise的解析。该函数带有单个参数：一个函数，对于方法什么时候可以继续，提供了值为`true`或者`false`的响应。我们把它设置为最多等待.5秒：

```
describe('googleApi', function() {
  beforeEach(function() {
    // 当我们等待
    // resolvedValue被解析的时候
    // 暂停本细则半秒
    waitsFor(function() {
      return resolvedValue !== undefined;
    }, 500);
  });

  it('should have a gapi function', function() {
    expect(
      typeof(googleApi.gapi)
    ).toEqual('function');
  });
```

现在我们可以为服务的返回值设置预期，并且断言它们等同于我们的设想：

```
it('should call gapi', function() {
  expect(googleApi.gapi.callCount)
    .toEqual(1);
});

it('should resolve with the browser keys', function() {
  expect(resolvedValue.clientId)
    .toBeDefined();
});
```

2. 端到端测试服务

既然服务是通过控制器与前端交互的，通过端到端测试来专门测试服务就不是很有效了。然而，我们也可以测试：服务解析了它们的promise，并且就是这些结果填充了视图。

例如，可以测试一个服务填充了视图中的一个事件列表。在一个用于显示事件列表的/events页面，我们可以断言确实列出了符合预期的事件数：

```
beforeEach(function() {
  browser().navigateTo('/#/events');
});

it('should show 10 events', function() {
  expect(
    repeater('.event_listing li').count()
  ).toBe(10);
});
```

19.14.5　测试过滤器

过滤器很容易测试：它们是隔离的功能。过滤器的功能就是限制或者改变输出，所以我们会在过滤器函数的输出上设置断言。

1. 单元测试过滤器

对过滤器进行单元测试很简单。首先，要访问过滤器，只需简单地把$filter服务注入到我们的测试中。这样我们就得到了一个在此过程中查找过滤器的途径：

```
describe('Unit: Filter tests', function() {
   var filter;

    // 在测试中模拟我们的引用
    beforeEach(module('myApp'));
    beforeEach(inject(function($filter) {
       filter = $filter;
   }));
});
```

有了对控制器的访问，在过滤器的输出上设置预期就是很容易的事了。

```
it('should give us two decimal points',
    function() {
        expect(filter('number')(123, 2)).toEqual('123.00');
});
```

2. 端到端测试过滤器

我们也可以用端到端测试在视图中测试过滤器的输出。端到端测试跟用于测试代码的单元测试略有不同，因为我们关注最终用户看到的东西，更甚于关注过滤器函数的具体输出。

要建立一个过滤器测试，需要让浏览器加载用于测试过滤器的页面，然后再跟过滤器自身打交道：

例如，给定如下用例，有一个对ng-repeat的实时搜索：

```
<input ng-model="search.$" type="text" placeholder="Search filter" />
<table id="emailTable">
    <tbody>
        <tr ng-repeat="email in emails | filter:search.$">
            <td>{{ $index + 1 }}</td>
            <td>{{ email.from }}</td>
            <td>{{ email.subject | capitalize }}</td>
```

```
        </tr>
    </tbody>
</table>
```

我们的emails数据看起来是下面这样的：

```
[
    {
       from: 'ari@fullstack.io',
        subject: 'ng-book and things'
    },
    {
        from: 'ari@fullstack.io',
        subject: 'Other things about ng-book and angular'
    },
   {
       from: 'ted@google.com',
       subject: 'Conference speaking gig'
   }
];
```

在此，我们要测试两个过滤器：在ng-repeat上设置了预期的那个实时搜索过滤器，以及在subject上的（确保它是大写的）过滤器。

要确保实时搜索是起作用的，需要在输入字段中输入一个值，并且确保这个字段按照我们的预期来变化。

```
it('should filter on the search', function() {
    expect(repeater('#emailTable tbody tr')).count()
        .toBe(3);
    // things过滤出email中的两条数据
    input('search.$').enter('things');
    expect(repeater('#emailTable tbody tr')).count()
        .toBe(2);
});
```

可以看到当给search.$字段设置了不同输入时，ng-repeat从3变成了2。

也可以测试我们自己的过滤器（大写过滤器），确保它是按照预期方式工作的：

```
it('should capitalize the subject line', function() {
    expect(
    repeater('#emailTable tbody tr:first')
        column('email.subject')
    ).toEqual(["Ng-book and things"]);
});
```

19.14.6　测试模板

测试模板时，我们着重于确保：特定的内容模板被加载了，模板中特定的数据也在视图中显示了。

1. 单元测试模板

既然模板是直接绑定到视图的，那么在视图中对组件的单元测试也就没有什么意义了：我们断言的建立基础是，视图的创建依赖于多个组件的完成。

可以建立一个断言：模板正常加载了。要做到这个，需要建立测试来期望对主页模板的一个

请求，并且执行一个视图的变更来验证它是不是真的加载了。

```
describe('Unit: Templates', function() {
    var $httpBackend,
        location,
        route,
        rootScope;

    beforeEach(module('myApp'));
    beforeEach(inject(
        function(_$rootScope_, _$route_, _$httpBackend_, _$location_){
          location = _$location_;
           rootScope = _$rootScope_;
           route = _$route_;
         $httpBackend = _$httpBackend_;
   }));

   afterEach(function() {
       $httpBackend.verifyNoOutstandingExpectation();
       $httpBackend.verifyNoOutstandingRequest();
   });

   // 我们的测试代码放在这里
});
```

现在，可以建立测试来反映导航到应用不同部分时的预期。

```
it('loads the home template at /', function() {
    $httpBackend.expectGET('templates/home.html')
        .respond(200);
    location.path('/');
    rootScope.$digest(); // 调用digest循环
    $httpBackend.flush();
});

it('loads the dashboard template at /dashboard', function() {
   $httpBackend.expectGET('templates/dashboard.html')
       .respond(200);
   location.path('/dashboard');
   rootScope.$digest(); // 调用digest循环
   $httpBackend.flush();
});
```

注意，我们并未在测试中返回一个模板（是.respond(200)，而不是.respond(200, "\<div\>\</div\>")）。鉴于我们只是在验证模板是否在请求中加载了，没有必要担心到底显示了什么。

端到端测试才是我们要验证视图外观是否与预期相符的地方。

2. 端到端测试模板

当在端到端测试中测试视图时，相比模板是否加载了，我们更加关注视图上加载的真实数据。通过这些测试，我们能够更好地了解用户在视图加载时看到的实际内容。

当它在用户的视图中时，我们将测试视图的内容。

为了测试模板，我们将测试视图包含了期望的HTML，还有它并未加载应用的其他部分。

```
describe('E2E: Views', function() {
    beforeEach(function() {
        browser().navigateTo('#/');
```

```
    });

    it('should load the home template', function()
    {
        expect(
            element('#emailTable').html()
        ).toContain('tbody');
    });

    it('should not load the dashboard template',
    function() {
        expect(
            element('#dashboard').count()
        ).toBe(0);
    });
});
```

对视图模板创建断言很简单。

19.14.7　测试指令

指令是Angular工作流程的根本。鉴于我们在Angular应用中做的绝大部分操作都是基于指令运作的，测试指令功能是最重要的组件测试之一，尤其是创建较大的组件时。

测试指令时，我们希望测试指令确实被加载到视图上了，并且也是按照我们期待它在DOM里$scope上的行为来运行的。

单元测试主要测试指令的功能，而在端到端测试里，可以检验是否在正确地使用指令。

1. 单元测试指令

单元测试指令需要建立一个测试来确保指令按照我们期望的那样来渲染。我们会要设置预期：绑定被正确建立了，错误按照期望的方式抛出了，视图按照预期显示了整个指令。

在这个测试里，我们会跟下面的指令打交道：

```
angular.module('myApp')
.directive('notification', function($timeout) {
    var html = '<div class="notification">' +
                 '<div class="notification-content">' +
                   '<p>{{ message }}</p>' +
                 '</div>' +
               '</div>';
    return {
        restrict: 'A',
        scope: { message: '=' },
        template: html,
        replace: true,
        link: function(scope, ele, attrs) {
            scope.$watch('message', function(n, o) {
                if (n)
                    $timeout(function() {
                        ele.addClass('ng-hide');
                    }, 2000);
            });
}}});
```

为了单元测试指令，我们需要把它们暴露到视图中。与测试控制器类似，我们需要手动把指

令放到一个待创建的元素中（在某种意义上与Angular第一次放置指令类似）。

```
describe('Unit: Directives', function() {
    var ele, scope;
    // 加载应用
    beforeEach(module('myApp'));
    // 我们的测试代码放在这里
});
```

现在，为了把指令加载到视图中，我们需要编译HTML内容，并且实施绑定，在生产中Angular会自动帮我们做这些。

```
describe('Unit: Directives', function() {
    var ele, scope;
    //加载应用
    beforeEach(module('myApp'));
    beforeEach(inject(function($compile, $rootScope) {
        scope = $rootScope;
        ele = angular.element(
            '<div notification message="note"></div>'
        );
       $compile(ele)(scope);
       scope.$apply();
   }));

  // 我们的测试代码放在这里
});
```

注意，我们在创建一个元素，这个元素调用了指令，就像我们直接把它放在DOM里那样。然后，需要编译这个元素，运行digest循环把它放在我们的假DOM上并且生效。

要测试这个指令，要做的就是与它交互，就好像它是在我们的DOM中一样。为了使指令上的绑定产生变化，需要强制运行一个digest循环。

在指令中，我们添加了一个到scope的message属性的绑定，所以当我们改变它时，需要在$apply()中调用：

```
// ……测试应用
it('should display the welcome text', function() {
    scope.$apply(function() {
        scope.note = "Notification message";
});

expect(
    ele.html()
    ).toContain("Notification message");
});
```

那么，不直接把HTML嵌入到指令的定义中，而是使用templateUrl选项来测试指令，又会怎样呢？在单元测试中，我们并不想建立真正的XHR请求来获取远程的模板。（Angular也根本就没有一个用于在测试中获取远程服务的方法。）建立真正的XHR请求会减慢我们的单元测试，并且让单元测试依赖于外部数据。如果外部数据源介入了测试功能，我们就没法知道测试在哪出错了。是外部数据源出错了吗？还是我们的功能出错了？

既然我们知道了XHR测试不能建立请求，一切需要通过templateUrl从远程地址加载模板的指令都会失败，除非在开始测试指令之前就先处理请求。幸好，Angular包含了$templateCache服务。

$templateCache服务允许$http服务缓存经过XHR的模板请求，这样它们就只会被请求一次。当一个模板被取到了，它的内容就会存储在$templateCache中，用模板路径作键。例如，当获取下面的实例指令时，它会请求templateUrl并且把模板的内容放在$templateCache中：

```
angular.module('myApp')
    .directive('notification', function($timeout) {
        return {
            restrict: 'A',
            scope: { ngModel: '=' },
            templateUrl: 'views/templates/notification.html',
}});
```

$templateCache 会 把 这 个 模 板 的 内 容 保 持 在 $templateCache('views/templates/notification.html')中。如果已经预先在$templateCache中存放了测试所需的指令文件内容，就可以使用$templateCache来阻止在指令的单元测试中再产生请求。

可以使用优秀的karma-ng-html2js-preprocessor包来把模板转换成可在测试中使用的Angular模块。

karma-ng-html2js-preprocessor

首先，需要安装包：

```
$ npm install --save-dev karma-ng-html2js-preprocessor
```

做好之后，需要修改karma.conf.js来使用html2js预处理器。我们需要在preprocessors数组中添加html2js预处理器：

```
preprocessors: {
    'app/views/**/*.html': ['ng-html2js']
},
// ...
```

为了让所有模板都被karma加载，需要设置files数组来加载模板。例如，如果模板放在app/view/下面，可以把这段加到files数组中以加载它们（根据具体项目来修改这个路径）：

```
files: [
    'app/bower_components/angular/angular.js',
    // ...
    'app/views/**/*.html',
    'app/scripts/*.js',
    // ...
]
```

围绕预处理器，需要加一些配置。我们会要创建用于注入到测试中的模块。此外，也会要把模板的cacheId设置为模板路由的真实文件路径。ng-html2js包允许我们为每个模板定义它，它存放在files数组中。

如果我们从指令中获取模板，很可能不是指向app/路径的（可以是相对于app/目录的）。在这些情况下，我们会从这个路径剥离路径，这样就可以引用这个URL，因为它已经被指令自己请求过了。配置可能看上去像下面这样：

```
// ...
ngHtml2JsPreprocessor: {
    moduleName: 'templates',
```

```
        cacheIdFromPath: function(filepath) {
            return 'app/views/' + filepath;
        },
        stripPrefix: 'app/'
    },
```

在我们的测试里，可以接着把templates模块注入到测试中：

```
describe('Directive: myDirective', function () {

    // 加载指令的模块
    beforeEach(module('myApp'));
    beforeEach(module('templates'));
```

templates模块注入到测试中之后，模板就已经通过templateUrl加载到指令中去了。因为模板会被自动加载，无需再做什么来测试带有templateUrl的指令。

2. 端到端测试指令

当对指令进行端到端测试时，我们主要关心的并不是指令的功能：我们测试的是在一个全局视图里，用户看到了什么，做了什么。

既然只是在测试视图，指令的端到端测试看上去跟测试模板差不多。

```
describe('E2E: directives', function() {

    beforeEach(function() {
        browser().navigateTo('/');
    });

    it('should have the welcome message', function() {
        expect(
            element('.notification', 'Notification').html()
        ).toContain('Notification message');
    });
});
```

19.15 测试事件

在应用中使用事件，尤其是那些导致DOM产生交互的事件时，我们要建立测试来保证真实的事件被触发了，它也得到了期待的事件数据。

我们也想建立测试来为在浏览器中捕获事件设置预期，想看看我们对事件的交互是否产生了期望的响应。

1. 单元测试事件

当对事件的触发进行单元测试时，我们感兴趣的是它们究竟调用了什么，还有是否真的调用了正确的事件。其次，我们主要关心处理程序有它们需要的数据。

使用Jasmine的辅助方法spnOn()可以非常容易地建立事件测试。

设想我们在测试一个控制器，它触发了一个$emit函数。基于这个函数，我们可以建立一个预期：事件被触发了，并且用我们所感兴趣的任意参数调用了。

首先，跟往常一样，需要建立测试，这样我们可以访问控制器的作用域：

```
describe('myApp', function() {
    var scope;
    beforeEach(angular.mock.module('myApp'));
    beforeEach(angular.mock.inject(function($rootScope) {
        scope = $rootScope.$new();
    });
});
```

测试建好之后，就可以简单地在作用域上为$emit或者$broadcase事件设置一个spyOn()调用了。

```
 // ...
 });
 it('should have emit called', function() {
     spyOn(scope, "$emit");
     scope.closePanel(); // 示例
                         // 或者任意可能导致emit被调用的事件
     expect(scope.$emit)
    .toHaveBeenCalledWith("panel:closed",
        panel.id);
});
```

我们也可以测试事件：设置一个事件触发后调用的$on()监听器。要执行$broadcast方法，可以简单地在作用域上调用$broadcast，并且对事件将会导致的作用域变化建立一个预期。

```
// ...
it('should set the panel to closed state',
    function() {
        scope.$broadcast("panel:closed", 1);
        expect(scope.panel.state).toEqual("closed");
});
```

2. 端到端测试事件

要对事件调用进行端到端测试十分简单：只要简单地测试事件产生的功能是按照我们的预期来触发的。

19.16　对 Angular 的持续集成

Angular的Karma与持续集成服务协作得很好。持续集成服务，或者简称为CI，给了我们的应用一个开发的入口，让我们对每次提交的代码能按照预期运行有信心。

持续集成服务器在全世界大小公司和开发人员中广泛使用，学习如何建立它们会是一个不错的主意。JenkinsCI[①]和TravisCI[②]都能很容易跟Karma集成，我们强烈推荐读者使用它们。

19.17　Protractor

新的端到端测试框架称为Protractor，它将会最终取代当前框架，成为默认的框架。

① http://jenkins-ci.org

② travis-ci.org

不像Angular场景运行器，Protractor是建立在WebDriver[1]上的，它是一种作为扩展编写的、用于控制浏览器的API。

WebDriver拥有IE、Chrome、Safari、Firefox等浏览器的控制器扩展，给我们提供了更多选择以及大量测试用的浏览器。这有很多好处，包括更稳定、更快捷。

Protractor就像Angular场景运行器那样，把实现Jasmine作为自己的测试平台，所以我们无需为了用它而学习一个全新的测试框架。

分度器自身可以被安装为独立的运行器，也可以当做一个库嵌入到我们的测试中。

安装

可以使用npm来安装Protractor：

```
$ npm install -g protractor
```

-g参数告诉npm全局安装protrator。

不像Angular场景运行器那样，Protractor需要一个独立的服务器运行在http://location:4444（这个位置可以配置）。

幸运的是，Protractor自带了一个工具，简化了Selenium服务器的安装。

为了访问脚本，我们需要把Protractor安装在待测试的Angular应用的顶级目录。

```
$ npm install protractor
```

然后运行本地node_module/目录下的Seleniun安装脚本：

```
$ ./node_modules/protractor/bin/webdriver-manager update --chrome
```

这个脚本会下载运行Selenuim所需的文件，并且用它们创建一个启动脚本和一个目录，如图19-7所示。

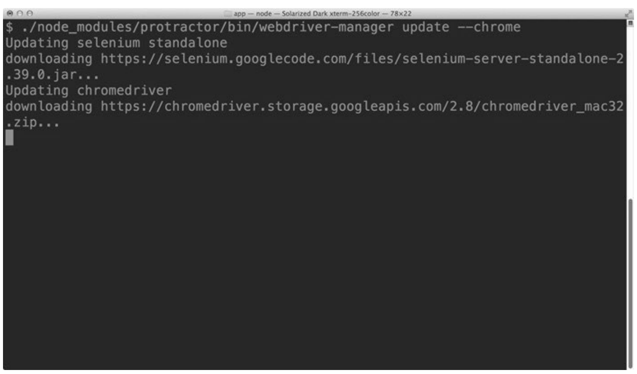


图19-7　安装Selenium

① https://code.google.com/p/selenium/wiki/WebDriverJS

这个脚本完成时，就可以执行这个启动脚本，用Chrome驱动来启动Selenuim的独立版本，如图19-8所示。

```
$ ./node_modules/protractor/bin/webdriver-manager start
```

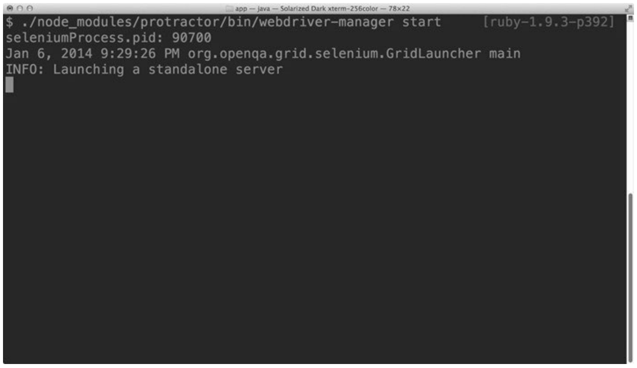


图19-8　安装Selenium

如果在安装Selenuim的过程中出现问题，试试从http://chromedriver.storage.googleapis.com/index.html下载并更新最新版本的ChromeDriver。

现在就可以用Protractor来连接在后台运行的Selenuim服务器了。

19.18　配置

就像Karma一样，Protractor需要运行一个配置脚本来告诉Protractor运行器如何连接到Selenuim，用哪个（哪些）浏览器，以及这些测试文件在哪里。

创建配置文件最简单的方式是从Protractor的安装中复制一个基准配置文件：

```
$ cp ./node_modules/protractor/example/chromeOnlyConf.js protractor_conf.js
```

为了让Protractor能运行，需要对这个脚本作一些修改。首先，默认配置脚本使用的Chrome驱动并不在我们的当前目录。我们需要把它指向本地./node_modules目录下的ChromeDriver。

```
chromeDriver: './node_modules/protractor/selenium/chromedriver',
```

然后，需要把specs数组指向我们的本地测试。

```
specs: ['test/e2e/**/*_spec.js'],
```

运行

我们有两个选项可以用来运行Protractor测试，第一个就是当运行Protractor时，使用Protractor来启动Selenium。这个选项叫做独立模式。我们复制的这个示例Protractor配置文件包含了这个设置。

```
chromeOnly: true,
chromeDriver: './node_modules/protractor/selenium/chromedriver',
```

运行Protractor测试的第二个方法是连接到一个单独运行的Selenuim服务器。当我们的测试变得更复杂时，会更希望在一个独立的Selenium服务器上运行我们的测试。

要把Protractor配置成使用这个独立的服务器，需要删除前面两个选项（`chromeOnly`和`chromeDriver`），并且添加`seleniumAddress`选项，指向运行的Selenium服务器。

```
seleniumAddress: 'http://0.0.0.0:4444/wd/hub',
```

19.19　配置选项

Protractor包含了多个选项，能让我们配置如何用我们的细则文件来运行Protractor。

1. `seleniumServerJar`（字符串）

把`seleniumServerJar`设置成独立Selenium服务器的路径，我们可以在运行测试时让Protractor启动Selenium服务器。

当我们运行持续集成（CI）测试时，这个设置很有用，但是Selenium的启动会很慢，所以不太适合在开发时使用。

2. `seleniumPort`（整数）

启动Selenium服务器的端口，如果Protractor要启动服务器的话。

3. `chromeDriver`（字符串）

当启动Selenium服务器时，作为`webdriver.chrome.driver`启动的ChromeDriver路径。如果这个变量为`null`，Protractor会尝试从`PATH`环境变量中寻找ChromeDriver。

4. `seleniumArgs`（数组）

这个数组中的字符串是我们可以在启动时手动传递给Selenium服务器的附加参数。

5. `sauceUser / sauceKey`（字符串）

如果设置了`sauceUser`和`sauceKey`，Selenium就不会启动，测试会在云测试服务SauceLabs[①]上远程运行。

6. `seleniumAddress`（字符串）

如果我们运行自己的服务器，这个字符串是运行中的Selenium服务器的地址。例如，如果我们用包含的脚本启动了Selenium服务器，这个字符串就会被设置成：http://localhost:4444/wd/hub 。

7. `allScriptsTimeout`（整数）

这个整数是每个脚本在浏览器中运行的超时时间。就是说，如果一个脚本运行了超过这个时间还没完成，就会被杀掉，然后报告为失败。

8. `specs`（数组）

这个字符串数组是要运行的细则的地址。它们可以是相对的、绝对的文件路径，或者是待匹

① https://saucelabs.com/

配的模式。

```
specs: [
    'spec/*_spec.js',
],
```

9. **capabilities**（对象）

这个对象包含了要传递给webdriver实例的兼容性键–值对。

```
capabilities: {
    'browserName': 'chrome'
}
```

10. **baseUrl**（字符串）

这个字符串是应用的基准URL。如果我们的测试使用了相对路径，这些路径就会被追加到这个字符串之后。

```
baseUrl: 'http://localhost:9000'
```

11. **rootElement** （字符串）

这个字符串是Angular应用寄生元素的选择器。默认为body，但如果我们的Angular应用包含在<body>元素的下级节点中，就需要包含这个选项。

12. **onPrepare**（字符串/函数）

在Protractor被启动并准备运行之后，所有细则被执行之前，这个函数会运行。它也可以是一个要在细则执行前运行的包含了代码的文件。onPrepare选项对于建立Jasmine报表是很有用的，例如：

```
onPrepare: function() {
    // 例如，添加一个Jasmine报表：
    jasmine.getEnv()
        .addReporter(new jasmine.JUnitXmlReporter(
            'outputdir/', true, true)
        );
},
```

13. **params**（对象）

params对象会被直接传递给Protractor实例，可以在我们的测试中访问这个对象。可以在这个params对象中存放任意值。

```
params: {
    env: 'test'
}
```

14. **jasmineNodeOpts**（对象）

jasmineNodeOpts指定了传递给JasmineNode实例的选项，jasmineNodeOpts的完整选项可以在https://github.com/juliemr/minijasminenode上找到。

一个完整的配置文件示例可能是这样的：

```
exports.config = {
  capabilities: {
      'browserName': 'chrome'
```

```
    },

    seleniumAddress: 'http://localhost:4444/wd/hub',

    baseUrl: 'http://localhost:9000',

    specs: ['test/protractor/**/*.js'],

    jasmineNodeOpts: {
        showColors: true,
        defaultTimeoutInterval: 30000
    }
};
```

19.20 编写测试

当用Protractor开始测试时，我们通过Jasmine来设置测试。也就是说，我们只是像编写Karma测试那样来编写测试。例如，一个简单的Protractor测试设置可能是这样的：

```
describe('homepage', function() {
    beforeEach(function() {
    // 函数执行之前
    });

    it('should load the page', function() {
    // 将测试代码放在这里
    expect(...).toEqual('hello');
  });
});
```

尽管上面测试的内容还没写完，但结构很熟悉了：它是用Jasmine建立的。我们使用`beforeEach()`和`afterEach()`函数以及嵌套的`describe()`块作为测试的结构。

要真正实现测试，使用Jasmine给予我们的同样的`expect()`语法。

编写Protractor测试需要我们在测试中跟Protractor暴露的一些全局变量打交道。下面列出了这些全局变量：

1. browser

`browser`变量是围绕webdriver实例的一个包装。我们使用`browser`变量来做各种导航，或者是从页面上抓取任意信息。

可以使用`browser`变量的`get()`函数来导航到一个页面：

```
beforeEach(function() {
    browser.get('http://127.0.0.1:9000/');
});
```

可以用`browser`对象来玩一些妙招。例如，可以用`browser`对象上的`debugger()`方法来调试页面。

```
it('should find title element', function() {
    browser.get('app/index.html');

    browser.debugger();
```

```
    element(by.binding('user.name'));
});
```

要把这个测试丢到node调试器里运行，可以在调试模式下运行测试：

```
$ protractor debug conf.js
```

在调试模式下运行Protractor时，我们得到的好处是让浏览器的执行停止，而且Protractor提供的每个客户端脚本在控制台中都可用。

要对Protractor的客户端脚本进行访问，我们可以用Protractor插入的`window.clientSideScripts`对象来调用它们。

2. element

`element`函数帮助我们找到正在测试的页面上的元素，并且与它们交互。

记住，`element`的返回值并不是一个DOM元素，它是`ElementFinder`的一个实例，`ElementFinder`是一个通过webdriver运作的对象。可以用这个对象的方法，如`sendKeys`和`click`，来跟页面上的对象交互。

完整的API太多了，更多的文档可以在Github上的文档页找到：https://github.com/angular/protractor/blob/master/docs/api.md

3. by

这个选项是元素定位策略的集合。可以通过CSS选择器、ID或者甚至绑定了`ng-model`的属性用它来查找元素。

by.binding　利用这个选项能可以查找使用`ng-bind`或者模板标记`{{ }}`绑定的元素，比如：

```
<h2 ng-bind="person.name"></h2>
```

我们可以在测试中使用下面这句来定位`<h2>`元素：

```
element(by.binding('person.name'))
```

by.model　可以使用`by.model`来搜索使用`ng-model`绑定的输入元素，比如：

```
<div my-directive ng-model="user.name"></div>
```

可以使用这句来定位`<div>`指令：

```
element(by.model('user.name'))
```

by.repeater　可以搜索包含了`ng-repeat`指令的元素。

比如，如果我们在一个`ng-repeat`中循环一个用户列表：

```
<ul>
    <li ng-repeat="user in users"></li>
</ul>
```

可以定位`<li>`指令，用下面的方式使用`by.repeater`来获取一个行的列表：

```
element.all(by.repeater('user in users'))
```

by.id by.id使我们能够通过CSS ID来搜索元素。例如，如果我们有一个带有如下id的<div>：

```
<div id="welcome_msg"></div>
```

可以通过如下代码来定位<div>：

```
element(by.id('welcome_msg'))
```

by.css 可以使用by.css来通过CSS选择符搜索元素。鉴于我们会经常使用CSS选择符来选择，为方便起见，Protractor把全局变量$绑定到了函数element.by.css上。

带有如下HTML：

```
<div class="container">
    <h2>Header</h2>
</div>
```

可以用以下代码来定位<h2>标题：

```
element(by.css('.container h2'))
```

4. **protractor**

这个变量是从webdriver转来的Protractor命名空间。它包含静态变量和类，可以用来操作Seleniun运行的浏览器的DOM实例。把这些全局变量组合成分度器实例，我们就能写出更高效的端对端测试。

> Protractor假定我们在测试Angular应用，并且期望页面上已有Angular。如果没有发现，会抛出一个错误。使用browser.driver对象来深入底层的webdriver实例，通过这种方式可以加载非Angular应用，不过这个已经超出本书范围了，此处不再讨论。

例如，我们要测试AngularJS首页上的基础示例：

```
describe('angularjs homepage', function() {
  it('should greet the named user', function() {
      // 加载AngularJS首页
  browser.get('http://www.angularjs.org');

  element(by.model('yourName'))
      .sendKeys('Ari');

  var greeting =
    element(by.binding('yourName'));

  expect(greeting.getText())
     .toEqual('Hello Ari!');
   });
});
```

使用browser对象，我们可以取回AngularJS首页。我们从这里寻找通过ng-model指令绑定到一个<input>字段的yourName变量。找到这个元素后，就让Selenium在里面输入Ari。然后我们寻找yourName绑定的元素（这是包在{{ yourName }}模板标签里的元素），并且设置一个预期：它应当包含文本“Hello Ari!”。

19.21　测试实践

尽管对于如何使用Protractor说起来很容易，关于怎么去做却并不那么明确。鉴于我们总是试着分享Angular上可用的最高质量的材料，我们会深入到对应用的测试和策略里。

19.21.1　我们的应用

设想我们有一个应用，为查看Github的问题提供了一个可选的视图。这个简单的应用本身只有几个主要特性：

- 允许用户在一个输入框中输入仓库的所有者和URL；
- 有一个主页，一个关于页；
- 它逐个列出了问题，包含用户的头像。

做好之后的应用如图19-9和图19-10所示。

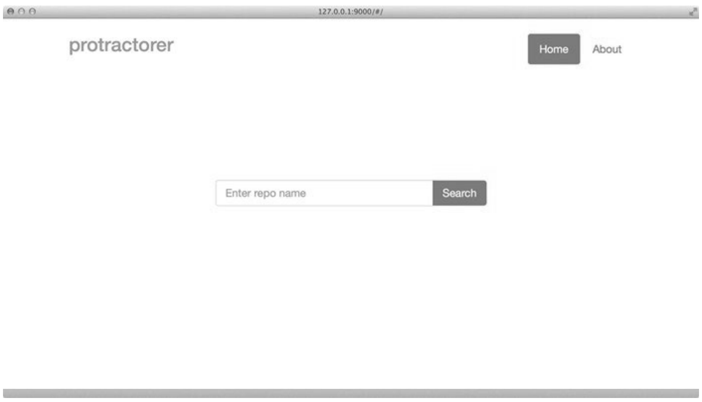


图19-9　最终的应用外观

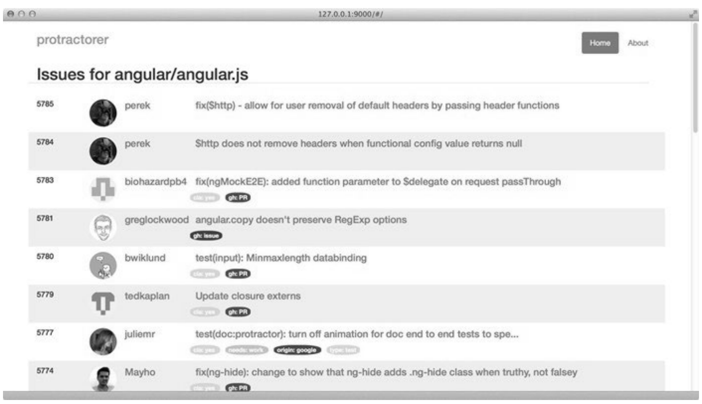


图19-10　最终应用

对于任何应用而言，我们都可以对要写的测试做一些策略。可能会对一个很小的简单应用写上百个测试，或者也可能写得很少。要同时写应用和测试时，找到两者之间的平衡会给我们带来好处。

19.21.2　测试的策略

我们已经发现，介于编写测试和编写代码之间的最佳平衡更多在于：知道要测试什么，怎样测试。无论什么时候给代码写测试，都要把测试定位到所实现的行为上来。也就是说，无须编写一个测试，来确保在`<input>`字段中输入时，`<h1>`标签的内容会变化。举例来说，我们需要为实时搜索测试自定义过滤。

我们还发现，在做原型之前，提前做测试没什么好处。在原型阶段，我们很少写测试（如果有的话），因为我们还在评审应用的功能。然而，当应用开始增多时，写测试就是个好主意了，它可以确保应用的行为在生产中是按照我们的预期来表现的。

最后，我们要建立测试，每一块测试的东西要尽可能少。理想情况下，每个测试块应当包含不超过一个预期。

理论说得够多了，我们用一些策略来测试应用。

我们想要测试页面更新成正在测试的仓库的标题。Angular应用使用一个自定义服务来创建到github.com的`$http`请求。这个请求返回后，我们在前台页面填充剩余部分。

其次，我们想要测试页面导航和内容的变化。这个测试涉及对视图上导航按钮的点击来提示`$location`变化。

我们开始吧！

19.22　建立我们的第一个测试

我们的Protractor配置文件很简单，在Protractor自带示例配置文件的基础上几乎没有修改：

```
// 示例配置文件
exports.config = {
  seleniumAddress: 'http://0.0.0.0:4444/wd/hub',
  capabilities: { 'browserName': 'chrome' },
  specs: ['test/e2e/**/*.spec.js'],
  jasmineNodeOpts: {
     showColors: true,
     defaultTimeoutInterval: 30000
  }
};
```

我们会在test/e2e目录中编写自己的Protractor测试，正如在配置文件中用命名约定`[name].spec.js`指定的那样。我们来在test/e2e目录中创建第一个测试，名为main.spec.js。

鉴于Protractor测试就是简单的Jasmine测试，我们从一个简单的Jasmine桩开始：

```
//在test/e2e/main.spec.js中
describe('E2E: main page', function() {
    // 把测试代码放在这里
});
```

考虑到我们在用Jasmine写测试，所以可以使用`beforeEach`块来建立它们。我们也需要跟踪Protractor实例，所以创建一个叫做`ptor`的变量来保持它。对于这些测试中的每一个，我们都会使用`browser`对象导航到首页。

因为是在做端对端测试，我们需要有一个服务器来让端对端测试在上面运行。

```
describe('E2E: main page', function() {
    var ptor;

    beforeEach(function() {
        browser.get('http://127.0.0.1:9000/');
        ptor = protractor.getInstance();
    });

});
```

不是每次要测试一个页面，都要把测试指向完整路径，可以在Protractor配置文件里面设置baseUrl。对于本节的剩余部分，我们默认这个选项被设置成这样：

```
baseUrl: 'http://127.0.0.1:9000/',
```

我们第一个测试会简单地测试主页加载：可以测试页面上存在某个元素。既然主页包含了一个ID #home，可以编写一个期望来保证这个条件。

首先使用by.id()函数来找到我们关注的元素，用#main ID来定位这个<div>：

```
it('should load the home page', function() {
    var ele = by.id('home');
});
```

有了这个元素以后，就可以使用Protractor的实例方法isElementPresent()来设置一个预期：元素在页面上。

```
it('should load the home page', function() {
    var ele = by.id('home');
    expect(browser.isElementPresent(ele)).toBe(true);
});
```

要运行我们的测试，需要启动一个Selenium服务器。幸好，Protractor通过一个叫做webdriver-manager的内置工具把这个过程变得很容易。这个管理器默认就被包含在内了（正如我们在上面看到的）。我们来启动webdriver-manager：

```
$ ./node_modules/protractor/bin/webdriver-manager start
```

在一个新的shell里，我们要启动Protractor来真正运行我们的测试。分度器二进制文件带单个参数：配置文件。运行过程如图19-11所示。

```
$ ./node_modules/protractor/bin/protractor protractor_conf.js
```

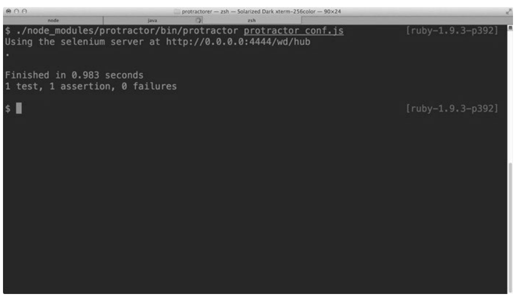


图19-11　运行过程

19.23　测试输入框

首先，我们把目光转向测试输入框。主页加载了一个表单，带有单个输入框，仅在用户未选择待搜索问题的仓库时显示，如图19-12所示。这个`<input type="text">`被绑定到称为`repoName`的模型。用户提交了表单以后，表单自身就消失了，取而代之的是问题列表显示。



图19-12　主页中有一个输入框

这段HTML如下所示：

```
<div id="repoform" class="main" ng-if="!repoName">
    <form ng-submit="getIssues()" class="input-group">
        <div class="input-group">
            <input type="text" ng-model='repo.name' placeholder='Enter repo name' />
            <span class="input-group-btn">
             <input type="submit" class="btn btn-primary" value="Search">
            </span>
        </div>
    </form>
</div>
```

我们在测试中所关注的功能是表单消失了，列表显示了。在一个新的测试中，我们要定位`<input>`元素，并且在里面写东西。可以用`sendKeys()`方法在目标元素上做这个。要定位`<input>`元素，可以使用`by.input()`方法找到含有`ng-model`绑定的输入元素。

```
it('the input box should go away on submit', function() {
    element(by.input('repo.name')).sendKeys('angular/angular.js\n');
});
```

我们运行这个测试时，会发现`<input>`字段被填充了。因为我们还没有写预期，所以也就没设置，但可以看到这个输入框被填入了angular/angular.js。

要让我们的输入框消失，需要提交表单。提交表单最简单的方法是假装按了回车按钮。在上面的`sendKeys()`中，我们包含了`\n`字符，它假装在`<input>`元素中按下了回车。

此时，我们只需建立一个预期：`repoform`元素不再在这个页面上存在了（因为我们用`ng-if`把它隐藏了）。可以用上面同样的方法来确认它不再存在于页面上：

```
it('the input box should go away on submit', function() {
    element(by.input('repo.name')).sendKeys('angular/angular.js\n');
    expect(ptor.isElementPresent(by.id('repoform'))).toBe(false);
});
```

19.23.1　测试列表

在<input>元素上建立测试之后，我们可以继续测试列表页的功能，如图19-13所示。

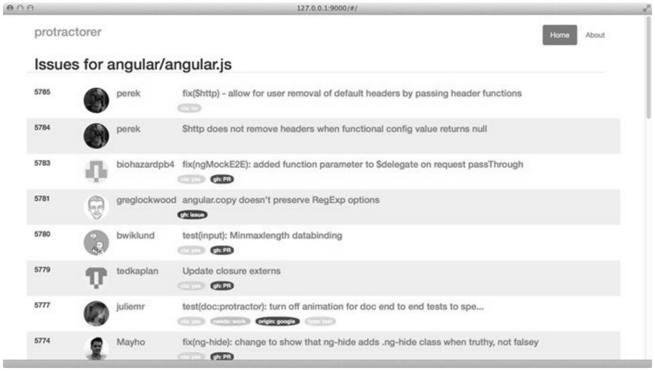


图19-13　测试列表页

既然剩下的待写测试都是在列表页中发生的，我们要把这里剩余的测试都嵌套在它们自己的describe()块中。使用一个独立的块能让我们设置另外一个beforeEach()块，在里面我们会建立针对angular/angular.js github仓库的测试。

这个describe块只会充当访问我们主页的用户，它在输入框中输入，然后按了回车。我们现行的测试方式似乎有些多余，但是请记住，端对端测试的目的是自动化用户交互。

```
describe('listing page', function() {
  beforeEach(function() {
      element(by.input('repo.name')).sendKeys('angular/angular.js\n');
  });
  // ...
  // 列表页测试会放在这里
});
```

列表页会有一些元素，通过ng-repeat迭代出来。使用GitHub API，会默认收到30个问题。我们可以测试这个页面确实解析了30个问题。

可以使用.repeater()帮助函数来定位ng-repeat元素、这个帮助函数在页面上寻找ng-repeat指令，并且找出匹配我们表达式的那一个。在这种情况下，我们是在使用Angular表达式d in data | orderBy:created_at:false 。可以用这句来定位循环器：

```
by.repeater('d in data | orderBy:created_at:false')
```

我们可以显式设置过滤器（就像上面那样），或者是停止这个过滤器，使用更通用的：

```
by.repeater('d in data');
```

使用by.repeater()并未真的返回任何元素，只是得到了一个用于获取元素的方法的指针。如果我们尝试对by.repeater()方法返回的对象设置期望，只会得到一个难看的错误。Protractor这么做是因为元素是通过promise填入的，所以我们要使用element.all()函数来获取解析后的元素。

```
var elems = element.all(by.repeater('d in data'));
```

定位元素之后，就可以在element.all()对象上要求一个总量，并且设置一个有30个元素的预期：

```
it('should have 30 issues', function() {
    var elems = element.all(by.repeater('d in data'));
    expect(elems.count()).toBe(30);
});
```

太棒了。我们来更深入了解这些重复的元素，确保每个元素都能显示头像。可以作一个合理的假设：每个元素都是由多个其他元素重复构成的，所以我们会建立一个测试来测试单个元素。

为了取得页面上的元素，我们会使用.repeater()方法，以同样的方式开始。element.all()方法返回一个对象，包含了多个方法，可以用来与重复的列表元素交互。在这种情况下，我们只使用first()方法来找到列表中的第一个元素。

鉴于列表还没有在页面上展示，first()方法返回一个promise，会在页面上解析为第一个元素。

```
it('includes a user gravatar per-element', function() {
    var elems = element.all(by.repeater('d in data'));
    elems.first().then(function(elm) {
        // elm就是第一个元素
    });
});
```

鉴于我们只对一个子元素感兴趣，所以会使用findElement()方法取得<img>元素。可以用多种方式定位这个元素，我们会使用.tagName()方法。跟first()方法一样，findElement()也因为同样的原因返回了一个promise。

```
it('includes a user gravatar per-element', function() {
    var elems = element.all(by.repeater('d in data'));
    elems.first().then(function(elm) {
        elm.findElement(by.tagName('img')).then(function(img) {
            // img就是<img>元素
        });
    });
});
```

我们特别感兴趣的是确保src属性包含了一个头像URL。可以使用element对象提供的各种方法去更加深入这个元素的细节。在本例中，我们会使用getAttribute()方法来找到src属性。跟前两个方法一样，也需要把它设置为promise：

```
it('includes a user gravatar per-element', function() {
    var elems = element.all(by.repeater('d in data'));
    elems.first().then(function(elm) {
        elm.findElement(by.tagName('img')).then(function(img) {
            img.getAttribute('src').then(function(src) {
                // src就是文本格式的img源地址
            });
        });
    });
});
```

鉴于我们得到src属性，所以可以建立一个期望：图片源与gravatar.com相匹配，如同GitHub使用Gravatar：

```
 it('includes a user gravatar per-element', function() {
     var elems = element.all(by.repeater('d in data'));
     elems.first().then(function(elm) {
         elm.findElement(by.tagName('img')).then(function(img) {
             img.getAttribute('src').then(function(src) {
                 expect(src).toMatch(/gravatar\.com\/avatar/);
             });
         })
     });
});
```

19.23.2　测试路由

要测试的最后一个功能是页面导航。正如预期的那样，我们会使用页面上的元素操作来建立测试。在本例中，我们会用CSS来定位/about链接，并且点击这个链接。

HTML代码如下所示：

```
 <div class="header">
   <ul class="nav">
     <li ng-class="{
       'active': isCurrentPage('')
     }"><a id="homelink" ng-href="#">Home</a></li>
     <li ng-class="{
       'active': isCurrentPage('about')
     }"><a id='aboutlink' ng-href="#/about">About</a></li>
   </ul>
  <h3 class="text-muted">protractorer</h3>
</div>
```

/about链接是header.nav列表中的第二个元素。定位这个列表最快的方法是通过by.css()方法使用CSS选择器。

```
it('should navigate to the /about page when clicking', function() {
    var link = element(by.css('.header ul li:nth-child(2)'))
});
```

因为有了这个链接，我们可以点击它，导航到新的URL。导航到/about页面后，就可以测试页面内容显示了，或者也可以测试当前的路由包含了/about路径。既然能合理地预期到Angular路由的工作，我们也可以默认：如果浏览器的URL匹配了about页，页面内容就会加载。

因此，我们只简单地测试当前URL包含/about。可以使用Protractor实例方法getCurrentUrl()来获得当前的URL：

```
it('should navigate to the /about page when clicking', function() {
    element(by.css('.header ul li:nth-child(2)')).click();
    expect(ptor.getCurrentUrl()).toMatch(/\/about/);
});
```

最后，既然我们测试的是前端，也要期望active样式类被加到链接上。

> active样式类在按钮上添加了颜色样式。

我们希望在点击/about链接时，执行和上面相同的操作。每当我们发现自己在复制测试数据时，就把测试嵌套在它们自己的describe块中，并且把副本移动到这个块中，通常这么做会比较

好。让我们继续吧，把测试移动到describe块：

```
describe('page navigation', function() {
    var link;
    beforeEach(function() {
        link = element(by.css('.header ul li:nth-child(2)'));
        link.click();
    });

    it('should navigate to the /about page when clicking', function() {
        expect(ptor.getCurrentUrl()).toMatch(/\/about/);
    });

    it('should add the active class when at /about', function() {
        // 应当有active样式类
    });
});
```

最后一个测试验证了class列表中是否包含字符串active：

```
expect(link.getAttribute('class')).toMatch(/active/);
```

19.24 页面对象

为了让我们的测试更加可读，可以使用webdriver的页面对象概念。页面对象基本上就是类，能让我们把特定的页面功能包装成一个干净的交互。

例如，可以把一个页面包装成这样：

```
var AngularHomepage = function() {
    this.nameInput = element(by.model('yourName'));
    this.greeting = element(by.binding('yourName'));

    this.get = function() {
        browser.get('http://www.angularjs.org');
    };

    this.setName = function(name) {
       this.nameInput.sendKeys(name);
    };
};
```

现在可以把测试整理成这样：

```
describe('angularjs homepage', function() {
    it('should greet the named user', function() {
        var angularHomepage = new AngularHomepage();
        angularHomepage.get();

        angularHomepage.setName('Julie');

        expect(angularHomepage.greeting.getText()).toEqual('Hello Julie!');
    });
});
```

页面对象允许我们创建对象，可以用在编写了很多测试的页面上。例如，可以创建一个LoginPage页面对象，把username和password字段包装成一个方法。我们可以简单地使用这个页

面对象方法来改变输入的值，而不是在每个测试中每次都手动选择`username`/`password`字段。

还要更多吗

Protractor是一个高度活跃的GitHub项目，也是个令人难以置信强大的端对端测试框架。它很快就会取代官方的端对端测试运行器Karma，并且会成为Angular官方的测试框架。

第 20 章 事件

20

在Web应用的组件是松耦合的情况下，比如需要用户验证然后处理授权，即时的通信不总是可行的，因为组件没有耦合在一起 。

例如，如果后端对一个请求返回了状态码401（表明一个未经授权的请求），我们期望Web应用不允许用户停留在当前视图，在这种情况下，我们希望应用把用户重定向到登录或者注册页面去。

基于这个逻辑，我们不能从外部告诉控制器设置一个新地址。我们也希望这个功能能覆盖多个作用域，这样可以用相同的行为来保护这些作用域。

我们需要另一种方式在它们之间通信。

Angular的作用域在本质上是分层次的：它们可以通过父子关系很自然地来回沟通。但通常，作用域是不共享变量的，它们执行的功能往往各不相同，跟在父树上的位置无关。

在这种情况下，我们可以通过在这个链上传递事件的方式在作用域之间通信。

20.1 什么是事件

如同浏览器响应浏览器层的事件，比如鼠标点击、页面滚动那样，Angular应用也可以响应Angular事件。这使我们可以在应用中嵌套的各组件之间进行通信，即使这些组件在创建时并未考虑到其他组件。

> 注意，Angular事件系统并不与浏览器的事件系统相通，这意味着，我们只能在作用域上监听Angular事件而不是DOM事件。

我们可以认为，事件是在应用中传播的信息片段，通常（可选）包含了在应用中发生的事情的信息。

20.2 事件传播

因为作用域是有层次的，所以我们可以在作用域链上传递事件。

通常来说，选择要使用的事件传递方式，一个好的经验法则是：查看将要触发事件的作用域。如果要通知整个事件系统（允许任意作用域处理这个事件），就要往下广播。

另一方面，如果要提醒一个全局模块（为了说），我们最终需要通知高层次的作用域（例如$rootScope），并且需要把事件向上传递。

> 限制向全局层面传递通知的数量是个好主意，尤其是因为事件虽然很强大，但增加了系统的复杂度。

比如，当我们在做路由的时候，"全局"应用状态需要知道应用当前设置了哪个页面。另一方面，如果我们是在一个选项卡指令和它的子面板指令之间通信，就需要把事件向下传。

20.2.1　使用$emit来冒泡事件

要把事件沿着作用域链向上派送（从子作用域到父作用域），我们要使用$emit()函数。

```
// 发送一个事件
// 我们的用户以当前user登录了
scope.$emit('user:logged_in', scope.user);
```

在一个$emit()事件函数的调用中，事件从子作用域冒泡到父作用域。在产生事件的作用域之上的所有作用域都会收到这个事件的通知。

当想要跟应用的其他部分交流状态的变更时，我们使用$emit()。如果想要跟$rootScope通信，需要$emit()这个事件。

$emit()方法带有两个参数。

1. name（字符串）

要发出的事件名称。

2. args（集合）

一个参数的集合，作为对象传递到事件监听器中。

$emit()方法返回了一个事件对象（关于事件对象的细节，查看20.3节）。

从监听器中发出的一切异常都会传递到$exceptionHandler服务中。

20.2.2　使用$broadcast向下传递事件

要把事件向下传递（从父作用域到子作用域），我们使用$broadcast()函数。

```
// 等等，购物车去结账了
// 当购物车在结账的时候
// 下面所有的指令都应当禁用自己
scope.$broadcast('cart:checking_out', scope.cart);
```

在$broadcast()方法上，每个注册了监听器的子作用域都会收到这个信息。事件传播到所有的指令和当前作用域的间接作用域上，并且一路往下调用每个监听器。

用了$broadcast()方法之后，就没法取消事件的发送了。

$broadcast()方法自身带有两个参数。

1. name（字符串）

要发出的事件名称。

2. args（集合）

一个参数的集合，作为对象传递到事件监听器中。

$emit()方法返回了一个事件对象（关于事件对象的细节，查看20.3节）。

从监听器中发出的一切异常都会传递到$exceptionHandler服务中。

20.3　事件监听

要监听一个事件，我们可以使用$on()方法。这个方法为具有某个特定名称的事件注册了一个监听器。事件名称就是在Angular中触发的事件类型。

例如，我们可以在路由变更过程被触发时，监听事件：

```
scope.$on('$routeChangeStart',
    function(evt, next, current) {
        // 一个新的路由被触发了
});
```

不管什么时候事件$routeChangeStart（路由将要变更的时候，会广播这个事件）被触发，监听器（这个函数）都会被调用。

Angular把evt对象作为第一个参数传给正在监听的一切事件，不管它是我们自定义的事件还是内置的Angular服务。

20.4　事件对象

事件对象有以下属性。

1. targetScope（作用域对象）

这个属性是发送或者广播事件的作用域。

2. currentScope（作用域对象）

这个对象包含了当前处理事件的作用域。

3. name（字符串）

这个字符串是触发之后，我们正在处理的事件名称。

4. stopPropagation（函数）

stopPropagation()函数取消通过$emit触发的事件的进一步传播。

5. preventDefault（函数）

preventDefault把defaultPrevented标志设置为true。尽管不能停止事件的传播，我们可以告诉子作用域无需处理这个事件（也就是说，可以安全地忽略它们）。

6. `defaultPrevented`（布尔值）

调用`preventDefault()`会把`defaultPrevented`设置为`true`。

`$on()`函数返回了一个反注册函数，我们可以调用它来取消监听器。

20.5　事件相关的核心服务

Angular核心框架发送事件，我们监听之后执行操作。可以用事件来让自己的Angular对象能在全局事件的不同状态上与应用交互。

我们用`$emit()`调用的有好几个事件，它们把事件往上发，更多调用的是`$broadcast()`事件。

20.5.1　核心系统的`$emitted`事件

下面的事件从指令向上发送到包含指令调用的作用域。我们可以使用`$on()`在这个链网上的任意作用域里监听这些方法：

```
$scope.$on('$includeContentLoaded',
    function(evt) {
});
```

1. `$includeContentLoaded`

`$includeContentLoaded`事件当`ngInclude`的内容重新加载时，从`ngInclude`指令上触发。

2. `$includeContentRequested`

`$includeContentRequested`事件从调用`ngInclude`的作用域上发送。每次`ngInclude`的内容被请求时，它都会被发送。

3. `$viewContentLoaded`

`$viewContentLoaded`事件每当`ngView`内容被重新加载时，从当前`ngView`作用域上发送。

20.5.2　核心系统的`$broadcast`事件

1. `$locationChangeStart`

当Angular从`$location`服务（通过`$location.path()`、`$location.search()`等）对浏览器的地址作更新时，会触发`$locationChangeStart`事件。

2. `$locationChangeSuccess`

当且仅当浏览器的地址成功变更，又没有阻止`$locationChangeStart`事件的情况下，`$locationChangeSuccess`事件会从`$rootScope`上广播出来。

3. `$routeChangeStart`

在路由变更发生之前，`$routeChangeStart`事件从`$rootScope`发送出来。也就是在路由服务开始解析路由变更所需的所有依赖项时。

这个过程通常涉及获取视图模板和解析route属性上所有依赖项的时候。

4. $routeChangeSuccess

在所有路由依赖项跟着$routeChangeStart被解析之后，$routeChangeSuccess被从$rootScope上广播出来。

ngView指令使用$routeChangeSuccess事件来获悉何时实例化控制器并渲染视图。

5. $routeChangeError

如果路由对象上任意的resolve属性被拒绝了，$routeChangeError就会被触发（比如它们失败了）。这个事件是从$rootScope上广播出来的。

6. $routeUpdate

如果$routeProvider上的reloadOnSearch属性被设置成false，并且使用了控制器的同一个实例，$routeUpdate事件会被从$rootScope上广播。

7. $destroy

在作用域被销毁之前，$destroy事件会在作用域上广播。这个顺序给予作用域一个机会，在父作用域被真正移除之前清理自身。

例如，如果我们在控制器中有一个正在运行的$timeout，我们不希望在包含它的控制器已经不存在的情况下，它还继续触发。

```
angular.module('myApp')
    .controller('MainController', function($scope, $timeout) {
        var timer;
        var updateTime = function() {
            $scope.date = new Date();
                timer = $timeout(updateTime, 1000);
            }
            // 开始更新时间
            timer = $timeout(updateTime, 1000);

            // 在销毁控制器之前
            // 清除定时器
            $scope.$on('$destroy', function() {
                if (timer) { $timeout.cancel(timer); }
            });
    });
```

第 21 章

架　构

学习Angular时，最令人迷惑的变化是要学习如何考虑应用的架构。尽管我们不能强制规定代码结构，因为那是开发人员的权利，但我们可以把我们的经验分享出来。

21.1　目录结构

AngularJS充满了各种可选项，Web应用的规模随着时间的推移而增长，所以很难决定如何组织代码。控制器放在哪里最合适？我们是应该把服务逻辑放在一个文件里，还是把它们拆散？

无论要构建什么规模的Angular应用，对结构方面的选择最好都要考虑一下：将要使用什么工具来构造这个应用，以及应用的大小。做的时候要带着这样的预期：项目是会增长的。

我们建议为应用程序创建以下目录结构，应用的文件放在/script目录，每个根据功能类型分开，另有一个总的app.js文件，如图21-1所示。

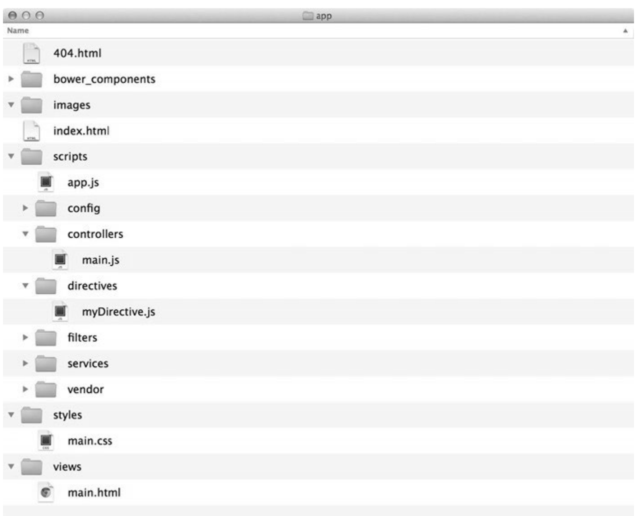


图21-1　推荐的目录结构

> 为了生产环境的需要，我们也建议使用一种像Grunt这样的工具把我们的文件合并成一个。

每个Angular对象都应当有自己的文件，根据其功能来命名。比如，`MainController`对象合

理的位置应该在scripts/controllers/main.js里，`myFilter`对象在scripts/filter/myFilter.js里。

采用这种结构是有好处的，因为每个文件都很小，根据功能区分，它能让多个开发人员有效地协同开发一个应用。

此外，我们建议为应用编写单元测试时，在根目录的test/下创建跟script目录同样的结构，如图21-2所示。

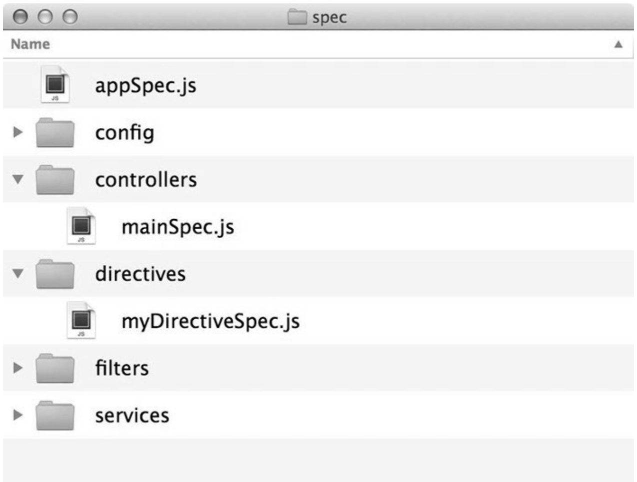


图21-2　推荐的测试目录结构

创建单模块应用时，我们建议用这样的结构。当创建多模块复合应用时，可以用一个有相似结构的module/目录作为顶层结构。

21.2　模块

模块是Angular应用的核心功能。模块包含我们为特定应用编写的所有代码，因而可能会快速膨胀。尽管这趋势也不算坏（模块是一种减少全局作用域干扰的好办法），我们还可以再细分模块。

关于何时创建模块，何时在全局模块中嵌套功能，有几种不同意见。下面两种方式都是把功能模块化的合理方式。

1. 函数模块化

最推荐的方法（我们以及Google团队）是根据功能将应用程序分解为模块。当根据功能分解应用程序时，我们打算聚焦于模块函数而不是包含的函数类型。

例如，当编写一个包含主页路由和登录路由的简单应用时，你可以构建两个模块来交付完全分离的功能。这个*主应用*看起来可能如下所示：

```
angular.module('app', [
    'app.home',
    'app:login'
]);
```

这个完整的应用程序被分解为两个分离的模块并且包含有限的功能，重点是粘合应用程序的不同部分。在这个案例中，我们有包含大多数逻辑的“home”界面和“login”界面。

以这种方式编写应用程序的好处是，我们可以在逻辑中分解函数的类型。你可以集中于为应用程序的每个组成部分提供一个个功能。还可以为我们编写的部分编写测试，从而分解测试。此外，还可以延迟加载应用程序的不同部分，因此可以为屏幕中最常见的界面提供最快的应用程序体验。

此外，这样做还有一个好处：一次编写模块就可跨应用程序共享它们。就是说，如果你的应用程序背后使用类似的登录系统，你也可以将这个应用程序的登录部分以及相关测试简单地复制、粘贴到另一个应用程序中。

在我们的产品应用程序中，我们使用这个方法编写模块化代码并且强烈推荐它。

2. 功能的模块化

要将我们的应用模块化，最显而易见的方式是按照模块功能划分。

我们需要把这些模块注入为主应用的依赖项，这样一来，为每个模块类型建立测试都非常容易，并且隔离和细分了功能，我们需要在编写细则时对这些功能作出解释。

比如，我们可以为每种Angular对象类型创建一个模块：

```
angular.module('myApp.directives', []);
angular.module('myApp.services', []);
angular.module('myApp.filters', []);
// 我们经常在控制器中使用服务，
// 所以，我们会把它们
// 注入到'myApp.controllers'模块中
angular.module('myApp.controllers', [
    'myApp.services'
]);
angular.module('myApp', [
    'myApp.directives',
    'myApp.controllers',
    'myApp.filters',
    'myApp.services'
]);
```

使用这种方法的一个问题是，它有时会给我们留下一堆小模块。这一结果不会影响性能，但会让开发变得繁琐。

21.3　控制器

当命名控制器时，惯例是使用控制器的名称，用大写字母开始，再加Controller。例如：

```
angular.module('myApp', [])
.controller('someController', function($scope) {
    // 一般不这么写
})
.controller('SomeController', function($scope) {
    // 这是控制器通常的写法
})
```

作用域的蔓延是Angular框架最令人困惑的一个方面，我们在控制器的$scope中定义了很多功能。编写Web应用时，有时会发现控制器的大小逐渐严重失控。

可以移出处理DOM的方法，以减少控制器的大小。把功能移动到自定义指令中，大幅降低了为判断是否公开特定视图或者格式化一个值的需要。

因为我们在视图里绑定了$scope上的值，控制器没有必要负责持有DOM对象所需的状态值。

比如，我们可以删除控制器中处理显示值的方法。

假设我们有一个登录页面，根据用户的选择的状态显示登录表单或者注册表单，把这个页面命名为showLoginForm：

```
angular.module('myApp', [])
.controller('LoginController', function($scope) {
    // 如果为true，显示为登录表单
    // 如果为false，显示为注册表单
    $scope.showLoginForm = true;
    $scope.sendLogin = function() {}
    $scope.sendRegister = function() {}
});
```

在我们的HTML里面，可能以如下方式设置LoginController的功能：

```
<div ng-show="showLoginForm">
    <form ng-submit="runLogin()"></form>
</div>
<div ng-show="!showLoginForm">
    <form ng-submit="runRegister()"></form>
</div>
```

尽管这个例子很普通（在我们的$scope里只有一个多余的变量），当我们的视图变得越来越复杂时，这种变量的数量会按指数级增长。

使用指令，我们可以把这个值去掉。例如：

```
angular.module('myApp', [])
    .directive('loginForm', function() {
        return {
            scope: {
                onLogin: '&',
                onRegister: '&'
            },
            templateUrl: '/templates/loginRegForms.html',
            link: function(scope, ele, attrs) {
                scope.showLoginForm = true;
                scope.submitLogin = function() {
                    scope.onLogin({user: scope.loginUser});
                }
                scope.submitRegister = function() {
                    scope.onRegister({user: scope.newUser});
                }
            }
        }
    });
angular.module('myApp', [])
    .controller('LoginController', function($scope) {
        $scope.sendLogin = function() {}
```

```
        $scope.sendRegister = function() {}
    });
```

我们可以在视图里调用这个指令，就像调用其他指令那样：

```
<div login-form
    on-login="sendLogin(user)"
    on-register="sendRegister(user)"></div>
```

我们的视图变量安全地藏到指令里去了，不再需要在控制器里持有视图条件了。让控制器精简是最佳实践，使用指令能让我们有效地做到这一点。

此外，像上面那样把登录的路由隔离到指令中，也使得测试它们的功能更容易。

在控制器之间共享数据

在Angular里，我们可以通过几种不同的方式在控制器之间传递数据。我们可以把控制器嵌套在同一个父控制器中，允许每个控制器独立修改父控制器$scope上的属性值，或者可以在服务中共享数据。

尽量把数据存在服务内部，但是更好的方式要视情况而定。比如说，在一个对话框里，把正在显示的数据放在父控制器里更有道理。

21.4　指令

知道什么时候写指令跟知道什么时候不写同样重要。更多的情况下，写指令是个好主意。

就像在上面看到的那样，它们减少了控制器之中的混乱。

测试指令也比测试控制器容易得多。

指令并不一定要有视图模板。通常情况下，它们可以只作为视图之下处理数据的垫片。ngModelController控制器就是这种功能派上用场的一个例子。

21.5　测试

我们总是鼓励测试我们的程序。我们不断为各种复杂度的功能编写单元测试。这些测试让我们对代码有信心，不管这信心有多小。专注于测试也让我们的时间更有效，并且更关注于功能。

只要确信已经对Angular应用API作过单元测试了，就可以编写端到端测试。端到端测试可以是脆弱的，依赖于视图，所以我们把它们留到开发过程的最后阶段。不仅如此，端到端测试一般比单元测试慢得多，所以，在开发过程中，先写单元测试能让我们更专注于功能的实现。

我们鼓励对应用的所有部分编写测试。关于测试的更多信息，请参阅第19章。

第 22 章

Angular动画

22

Angular团队创建了ngAnimate模块，让我们的Angular应用能够提供CSS和JavaScript动画。

在Angular应用中创建动画，有几种途径：

- 使用CSS3动画；
- 使用JavaScript动画；
- 使用CSS3过渡。

本章旨在讨论这三种动画方法，并且对如何能做出自己的动画有一个深刻的理解。

22.1 安装

自1.2.0起，动画就不再是Angular核心的一部分了，它们存在于自己的模块中。为了在Angular应用中包含动画，需要在应用中安装并且引用这个模块。

可以从code.angularjs.org[①]上下载它，并将它保存到一个能从HTML引用到的位置，比如js/vendor/angular-animate.js。

也可以使用Bower来安装它，这会把它放在Bower目录中。想要知道更多有关Bower的信息，请参阅34.6节。

```
$ bower install --save angular-animate
```

引用了Angular自身之后，需要在HTML中引用这个动画库。

```
<script src="js/vendor/angular.js"></script>
<script src="js/vendor/angular-animate.js"></script>
```

最后，需要在我们的应用模块中把ngAnimate模块当作依赖项来引用：

```
angular.module('myApp', ['ngAnimate']);
```

至此，就已经准备好使用AngularJS来呈现动画了。

22.2 它是如何运作的

$animate服务默认给动画元素的每个动画事件（参见后面的列表）添加了两个CSS类。

① http://code.angularjs.org

$animate服务支持多个Angular内置的指令，它们无需额外的配置即可支持动画。它很灵活，我们可以为自己的指令创建动画。

所有这些预先存在的支持动画的指令，都是通过监控指令上的事件实现的。例如，当一个新的ngView进入并且把新内容带进浏览器时，这个事件就叫做ngView的enter事件。当ngHide准备显示一个元素的时候，remove事件就会触发。

下面是指令以及在不同状态触发的事件列表。我们会使用这些事件来定义在每个状态上，动画会如何工作。

指　令	事　件
ngRepeat	enter、leave、move
ngView	enter、leave
ngInclude	enter、leave
ngSwitch	enter、leave
ngIf	enter、leave
ngClass或者class="…"	add、remove
ngShow	add、remove（.ng-class）
ngHide	add、remove

$animate服务基于指令发出的事件来添加特定的样式类。对于结构性的动画（比如进入、移动和离开），添加上去的CSS类是ng-[EVENT]和ng-[EVENT]-active这样的形式。

对于基于样式类的动画（比如ngClass），动画样式类的形式是[CLASS]-add、[CLASS]-add-actdive、[CLASS]-remove、[CLASS]-remove-active。

最后，对于ngShow和ngHide，只有.ng-hide类会被添加和移除，它的形式跟ngClass一样：.ng-hide-add、.ng-hide-add-active、.ng-hide-remove、.ng-hide-remove-active。

自动添加类

触发enter事件的指令会在DOM变更时收到一个.ng-enter样式类，然后，Angular添加ng-enter-active类，它会触发动画。ngAnimate自动检测CSS代码来判定动画什么时候完成。

这个事件完成时，Angular会从DOM元素上移除这两个类，使我们能够在DOM元素上定义动画相关的属性。

如果浏览器不支持CSS过渡或者动画，动画会开始，然后立即结束，DOM会处于最终的状态，不会添加过渡或者动画的样式类。

所有支持的结构性动画事件都遵循同样的约定：进入、离开、移动。基于样式的动画事件略有不同（如上所述）。

22.3　使用 CSS3 过渡

迄今为止，我们在应用中包含动画的最简单的方式是使用CSS3过渡，除了IE9和更早版本外，它都能运行。

不支持CSS3过渡的浏览器会优雅地降级到应用的无动画版本。

要做任何CSS动画，我们都要确认给动画中关注的DOM元素添加了样式。

例如，在下面的示例中，我们看看如何让这个元素动起来：

```
<div class="fade-in"></div>
```

CSS3过渡是完全基于样式类的，意思是说，只要我们在HTML上定义了动画的样式，这个动画就会在浏览器中动起来。

为了用样式类实现动画，需要遵循Angular的命名约定来定义CSS过渡。

CSS过渡是让元素从一种样式渐变为另一种样式的特效。要定义一个动画，我们需要指定想要添加动画的属性，以及特效的持续时间。

例如，这段代码使用`.fade-in`类在DOM元素的所有属性上添加了一个持续两秒的过渡特效。

```
.fade-in {
    transition: 2s linear all;
    -webkit-transition: 2s linear all;
}
```

设置了这个过渡和时间之后，就可以在DOM元素的不同状态上定义属性了。

```
.fade-in:hover {
    width: 300px;
    height: 300px;
}
```

使用`ngAnimate`，Angular通过给每个动画事件添加两个样式类的方式开始了我们的指令动画：初始的`ng-[EVENT]`类，不久之后是`ng-[EVENT]-active`类。

为了自动让上面的DOM元素使用过渡实现Angular动画，我们修改上面初始的`.fade-in`示例来包含初始状态类：

```
.fade-in.ng-enter {
    opacity: 0;
}
.fade-in.ng-enter.ng-enter-active {
    opacity: 1;
}
```

要真正运行这个动画，需要包含CSS动画定义。在这个定义中，我们需要同时包含持续时间和将要修改的元素属性。

```
.fade-in.ng-enter { transition: 2s linear all; -webkit-transition: 2s linear all; }
```

也可以把`transition`属性放到基准CSS类中（“`.fade-in`”，而不是在每个要产生动画的地方都指定）。

```
.fade-in {
    -webkit-transition: 2s linear all;
    transition: 2s linear all;
}
.fade-in.ng-enter {
    opacity: 0;
```

```
}
.fade-in.ng-enter.ng-enter-active {
    opacity: 1;
}
.fade-in.ng-leave {
    opacity: 1;
}
.fade-in.ng-leave.ng-leave-active {
    opacity: 0;
}
```

22.4　使用 CSS3 动画

CSS3动画比CSS3过渡更广泛、更复杂。除IE9和更早版本的IE之外，所有主流浏览器都支持它们。使用CSS3动画，我们会用同样的初始样式类ng-[EVENT]，但是不需要在ng-[EVENT]-active状态中定义动画状态，因为CSS规则会处理剩余部分。

我们在@keyframes规则中创建动画。在@keyframes规则中定义的CSS元素内部，我们定义要处理的CSS样式。

想让DOM元素动起来时，我们使用animation:属性来绑定@keyframe CSS属性，它把动画添加到CSS元素上。

当在CSS元素上绑定动画时，我们需要同时指定动画的名称和持续时间。

记得添加动画持续时间：如果我们忘记添加动画的持续时间，它默认会设成0，此时动画就不会运行了。

要创建@keyframes规则，我们需要给关键帧一个名字，并且设置动画的时间阶段，它包含了动画过程中的属性。

```
@keyframes firstAnimation {
    0% {
        color: yellow;
    }
    100% {
        color: black;
    }
}
/*对于Chrome和Safari浏览器 */
@-webkit-keyframes firstAnimation {
    /* from等于0% */
    from {
        color: yellow;
    }
    /* from等于100% */
    to {
        color: black;
    }
}
```

使用关键字from等同于把百分比设置为0%，使用关键字to等同于把百分比设置成100%。

我们并不局限于0%和100%：可以分步提供动画，比如10%、15%，等等。要把@keyframe属性赋值到想要应用动画的类上，我们使用animation关键字，它把动画应用到CSS选择器选定的元素上。

```
.fade-in:hover {
    -webkit-animation: 2s firstAnimation;
    animation: 2s firstAnimation;
}
```

用ngAnimate，我们把firstAnimation值绑定到任意用.fade-in类选定的元素上。Angular自动为我们添加和移除.ng-enter类，所以我们可以简单地把事件添加到.fade-in.ng-enter类上：

```
.fade-in.ng-enter {
    -webkit-animation: 2s firstAnimation;
    animation: 2s firstAnimation;
}
```

22.5 交错 CSS 过渡/动画

ngAnimate捆绑了一个额外的特性，用指定的延迟来间隔同时存在的动画。这意味着如果10个项进入了一个ngRepeat列表，每个项可以在上一个之后延迟X毫秒插入。这样产生的特效就是一个交错特效，ngAnimate把CSS过渡和动画处理成这样。

22.5.1 交错CSS过渡

沿用ng-enter和ng-enter-active这样组织CSS过渡代码的格式，可以添加一个额外的CSS类来提供交错延迟。使用下面的CSS代码，可以用CSS过渡来给我们的.fade-in类添加一个交错特效。

```
.fade-in.ng-enter-stagger {
    -webkit-transition-delay:200ms;
     transition-delay:200ms;

     /* 防止意外CSS继承的保护措施 */
     -webkit-transition-duration:0;
     transition-duration:0;
}
```

下面的代码会在每个后续项以动画方式进入之后，执行200毫秒的停顿。注意，另有一个CSS属性指定了持续时间，并且设置成零了。为什么？它在此是一个安全防护，防止意外的CSS继承基础CSS类。要是没有这种保障，交错特效可能就会被忽略了。

但是这对于我们的.fade-in类意味着什么呢？想象一下我们正在使用一个ngRepeat元素，这个元素使用的就是.fade-in类。

```
<div ng-repeat="item in items" class="fade-in">
```

```
    Item: #1 -- {{ item }}
</div>
```

每次一系列的项插入到列表中之后，交错延迟会逐步启动。Item #1会被正常插入，#1会在200毫秒之后，#3 400毫秒之后，以此类推。

22.5.2　交错CSS动画

CSS动画也支持并且遵循与上面提到的CSS过渡交错特效同样的CSS命名约定。唯一的不同是没有使用transition-delay，而是用了animation-delay（原因很明显）。如果用CSS动画来实现交错特效，.fade-in类看上去就会像这样：

```
.fade-in.ng-enter-stagger {
    -webkit-animation-delay:200ms;
    animation-delay:200ms;

    /* css交错动画需要放在这里 */
    -webkit-animation-duration:0;
    animation-duration:0;
}
```

既然CSS关键帧要等到重排（当浏览器重绘屏幕）时才会发生，可能会出现轻微的闪烁，或者元素自身可能短暂地不动，直到交错动画开始生效。这是因为关键帧动画尚未触发，所以from或者0%的动画还没有开始。为解决这个问题，在赋值了关键帧动画的CSS类中，可以放额外的CSS样式。

```
.fade-in.ng-enter {
    /* 重排之前的样式 */
    opacity:0;

    -webkit-animation: 2s firstAnimation;
    animation: 2s firstAnimation;
}

.fade-in.ng-enter-stagger { ... }

@keyframes firstAnimation { ... }
@-webkit-keyframes firstAnimation { ... }
```

22.5.3　什么指令支持交错动画

很简单，所有指令都可以，但是仅当同一父容器下的两个或更多相同动画事件同时触发时，才可以使用。所以当10个项被插入一个nRepeat列表时，交互特效就产生。这意味着如果ngClass被放在一个ngRepeat元素上，ngClass的值在列表中对每个项都产生了变化，样式类变化的动画就会渲染出一个交错特效。

交错动画也可以在自定义指令中触发。在一行中用$animate服务调几次，一个交互动画就呈现出来了。确保每个动画的父元素是同一个，并且每个参与动画元素的className值也是相同的。

22.6　使用 JavaScript 动画

JavaScrip动画不同于前两种Angular动画方法，因为我们直接使用JavaScript设置DOM元素的属性。

所有的主流浏览器都支持JavaScript动画，所以如果想在不支持CSS渐变和动画的浏览器上提供动画的话，这是个好的选择。

这里，我们更新JavaScript来处理动画，而不是操控CSS来让元素动起来。

ngAnimate在模块API上添加了.animation方法；这个方法提供了一个接口，我们可以用来创建动画。

animiation()方法带有两个参数。

- classname（字符串）

这个classname会匹配要产生动画的元素的class。到现在为止的例子里，这个动画应当被命名为：.fade-in。

- animateFun（函数）

animate函数预期会返回一个对象，包含了指令会触发的不同事件函数（当使用的时候）。

想知道这些函数的详细文档，请查阅$animate API文档。

```
angular.module('myApp', ['ngAnimate'])
    .animation('.fade-in', function() {
        return {
            enter: function(element, done) {
                // 运行动画
                // 当动画结束的时候调用done
                return function(cancelled) {
                    // 关闭或者取消的回调
                }
            }
        }
});
```

$animate服务为指定的元素调用这些函数。在这些函数里，我们可以对这个元素做任何事情。唯一要求是在动画结束时，需要调用回调函数done()。

在这些函数中，我们可以返回一个end函数，它会在动画结束或者动画被取消时调用。

当动画触发时，$animate为事件查找匹配的动画函数。如果找到了匹配事件的函数，它会执行这个函数，否则就会完全跳过这个动画。

22.7 微调动画

取决于应用的复杂度，ngAnimate和它下面的动画代码可能会需要一些调整。

对CSS类作过滤

默认情况下，ngAnimate会自动尝试让每个通过$animate服务传递过来的元素都动起来。但是不必担心，只有包含了用CSS或者JavaScript动画注册了的CSS类的元素才会真的动起来。

尽管这个系统在运作时，必须检查每个可能的CSS类，这可能会在低速设备上慢一些。因此，在angular.js的1.2.13发布版本中，ngAnimate提供了一个配置项。让$animate提供者可以使用正

则表达式对元素进行过滤，以去掉不匹配元素上的动画操作。

```
myModule.config(function($animateProvider) {
    // 唯一合法的参数是正则表达式
    $animateProvider.classNameFilter(/\banimate-/);
});
```

现在有了给定的正则表达式，/animated/，只有以animate开始的CSS类会被为动画而处理。结果，我们的.fade-in动画不会再运行了，它需要被重命名成.animate-fade-in才能真正运行。

22.8　DOM 回调事件

当动画在一个元素产生时，我们想要检测DOM操作什么时候发生，可以在$animate服务上注册一个事件。事件如下：

```
element.on('$animate:before', function(evt, animationDetails) {});
element.on('$animate:after', function(evt, animationDetails) {});
```

22.9　内置指令的动画

22.9.1　ngRepeat动画

ngRepeat指令产生这些事件：

动　作	事　件　名
一项被插入到列表之后	enter
一项从列表中移除	remove
列表中的一项移动了	move

对这三个例子，我们使用下面这个HTML：

```
<div ng-controller="HomeController">
  <ul>
    <li class="fade-in" ng-repeat="r in roommates">
      {{ r }}
    </li></ul>
</div>
```

我们的HomeController默认是这样定义的：

```
angular.module('myApp', ['ngAnimate'])
    .controller('HomeController', function($scope) {
        $scope.roommates = [
            'Ari', 'Q', 'Sean', 'Anand'
        ];
        setTimeout(function() {
            $scope.roommates.push('Ginger');
            $scope.$apply(); // 触发一次digest

            setTimeout(function() {
                $scope.roommates.shift();
                $scope.$apply(); // 触发digest
```

```
            }, 2000);
        }, 1000);
});
```

在这些例子中，我们有一个roommates列表，包含了四个元素。在一秒钟之后，加了第五个。两秒之后，移除了第一个元素。

1. CSS3过渡

要让ngRepeat列表中的元素动起来，我们需要确认添加了展现元素初始状态的CSS样式类，以及为enter和edit状态定义最终状态的类。

首先，在初始类上定义动画属性：

```
.fade-in.ng-enter,
.fade-in.ng-leave {
    transition: 2s linear all;
    -webkit-transition: 2s linear all;
}
```

至此，可以简单地在动画中定义初始和最终阶段的CSS属性。这里，我们把元素从绿色的文字淡入，在进入动画的最终阶段把文字变成黑色。在离开（元素移除）动画中，我们把属性反转：

```
.fade-in.ng-enter {
    opacity: 0;
    color: green;
}
.fade-in.ng-enter.ng-enter-active {
    opacity: 1;
    color: black;
}
.fade-in.ng-leave {}
.fade-in.ng-leave.ng-leave-active {
    opacity: 0;
}
```

2. CSS3关键帧动画

使用关键帧动画时，无需定义开始和结束的样式类，而是仅定义单个选择器，包含动画样式的键。

首先为关键帧定义动画属性：

```
@keyframes animateView-enter {
    from {opacity:0;}
    to {opacity:1;}
}
@-webkit-keyframes animateView-enter {
    from {opacity:0;}
    to {opacity:1;}
}
@keyframes animateView-leave {
    from {opacity: 1;}
    to {opacity: 0;}
}
@-webkit-keyframes animateView-leave {
    from {opacity: 1;}
    to {opacity: 0;}
}
```

设置了关键帧之后，我们可以简单地把动画附加到ngAnimate添加的CSS样式类上：

```
.fade-in.ng-enter {
    -webkit-animation: 2s fade-in-enter-animation;
    animation: 2s fade-in-enter-animation;
}
.fade-in.ng-leave {
    -webkit-animation: 2s fade-in-leave-animation;
    animation: 2s fade-in-leave-animation;
}
```

3. JavaScript动画

当用JavaScript做动画时，需要在动画的描述对象上定义enter和leave属性。

```
angular.module('myApp', ['ngAnimate'])
    .animation('.fade-in', function() {
        return {
            enter: function(element, done) {
                // 不使用jQuery的原始动画
                // 用jQuery会简单很多
                var op = 0, timeout,
                animateFn = function() {
                    op += 10;
                    element.css('opacity', op/100);
                    if (op >= 100) {
                        clearInterval(timeout);
                        done();
                    }
                };

                // 把初始透明度设为0
                element.css('opacity', 0);
                timeout = setInterval(animateFn, 100);
            },
            leave: function(element, done) {
                var op = 100,
                timeout,
                animateFn = function() {
                    op-=10;
                    element.css('opacity', op/100);
                    if (op <= 0) {
                        clearInterval(timeout);
                        done();
                    }
                };
                element.css('opacity', 100);
                timeout = setInterval(animateFn, 100);
            }
        }
});
```

22.9.2　ngView动画

ngView指令触发这些事件：

动　　作	事件名称
新的视图内容准备好了	enter
正在离开，已有的内容将被隐藏	leave

对这三个例子，我们使用下面这个HTML来运行：

```
<a href="#/">Home</a>
<a href="#/two">Second view</a>
<a href="#/three">Third view</a>
<div class="animateView" ng-view></div>
```

当跟ng-view指令协作时，我们是在跟Angular内部的路由打交道。有关路由的更多信息，请参阅第12章。Angular路由的下载地址是：http://code.angularjs.org/1.2.13/angular-route.js。

对于下面的示例，可以把路由设置为：

```
angular.module('myApp', ['ngAnimate', 'ngRoute'])
    .config(function($routeProvider) {
        $routeProvider.when('/', {
            template: '<h2>One</h2>'
        }).when('/two', {
            template: '<h2>Two</h2>'
        }).when('/three', {
            template: '<h2>Three</h2>'
        });
})
```

示例中的三个路由，每个显示了一个不同的视图。

1. CSS3过渡

要让ngView列表中的元素动起来，我们需要确认添加了展现元素初始状态的CSS样式类，以及为enter和edit状态定义最终状态的类：

```
.animateView.ng-enter,
.animateView.ng-leave {
    transition: 2s linear all;
    -webkit-transition: 2s linear all;
}
```

至此，可以简单地在动画中定义初始和最终阶段的CSS属性。这里，我们把元素从绿色的文字淡入，在进入动画的最终阶段把文字变成黑色。在离开（元素移除）动画中，我们把属性反转：

```
.animateView.ng-enter {
    opacity: 0;
    color: green;
}
.animateView.ng-enter.ng-enter-active {
    opacity: 1;
    color: black;
}
.animateView.ng-leave {}
.animateView.ng-leave.ng-leave-active {
    opacity: 0;
}
```

2. CSS3关键帧动画

首先，添加我们为动画定义的@keyframe：

```
@keyframes animateView-enter {
    from {opacity:0;}
    to {opacity:1;}
```

```
}
@-webkit-keyframes animateView-enter {
    from {opacity:0;}
    to {opacity:1;}
}
@keyframes animateView-leave {
    from {opacity: 1;}
    to {opacity: 0;}
}
@-webkit-keyframes animateView-leave {
    from {opacity: 1;}
    to {opacity: 0;}
}
```

为了应用动画，需要做的就是在我们的类中添加动画CSS样式：

```
.animateView.ng-enter {
    -webkit-animation: 2s animateView-enter;
    animation: 2s animateView-enter;
}
.animateView.ng-leave {
    -webkit-animation: 2s animateView-leave;
    animation: 2s animateView-leave;
}
```

3. JavaScript动画

首先，我们需要下载[1]并且在文档的头部包含jQuery。

当用JavaScript做动画时，需要在动画的描述对象上定义`enter`和`leave`属性。

```
angular.module('myApp', ['ngAnimate'])
    .animation('.animateView', function() {
        return {
            enter: function(element, done) {
                // 显示如何用jQuery实现动画的例子
                // 注意，这需要在HTML中包含jQuery
                $(element).css({
                    opacity: 0
                });
                $(element).animate({
                    opacity: 1
                }, done);
            },
            leave: function(element, done) {
                done();
            }
        }
});
```

22.9.3　ngInclude动画

`ngInclude`指令触发这些事件：

动　　作	事件名称
新的视图内容准备好了	`enter`
正在离开，已有的内容将被隐藏	`leave`

[1] http://jquery.com/download/

对这三个例子，我们使用下面这个HTML来运行：

```
<div ng-init="template.url='/home.html'"
    ng-controller="HomeController">
    <button ng-click="template.url='/home.html'">
        Home
    </button>
    <button ng-click="template.url='/second.html'">
        Second
    </button>
    <button ng-click="template.url='/third.html'">
        Third
    </button>
    <div class="animateInclude"
          ng-include="template.url">
    </div>
</div>
```

我们也在页面中包含内联模板（为了演示），也可以把这些视图设置为从远程服务器获取。

```
<script type="text/ng-template" id="/home.html">
    Home Template
</script>
<script type="text/ng-template" id="/second.html">
    Second Template
</script>
<script type="text/ng-template" id="/third.html">
    Third Template
</script>
```

1. CSS3过渡

要让ngInclude列表中的元素动起来，我们需要确认添加了展现元素初始状态的CSS样式类，以及为enter和edit状态定义最终状态的类：

```
.animateInclude.ng-enter,
.animateInclude.ng-leave {
    transition: 2s linear all;
    -webkit-transition: 2s linear all;
}
```

至此，可以简单地在动画中定义初始和最终阶段的CSS属性。这里，我们把元素从绿色的文字淡入，在进入动画的最终阶段把文字变成黑色。在离开（元素移除）动画中，我们把属性反转：

```
.animateInclude.ng-enter {
    opacity: 0;
    color: green;
}
.animateInclude.ng-enter.ng-enter-active {
    opacity: 1;
    color: black;
}
.animateInclude.ng-leave {}
.animateInclude.ng-leave.ng-leave-active {
    opacity: 0;
}
```

2. CSS3动画

首先，添加为动画定义的@keyframe：

```
@keyframes animateInclude-enter {
    from {opacity:0;}
    to {opacity:1; color: green}
}
@-webkit-keyframes animateInclude-enter {
    from {opacity:0;}
    to {opacity:1; color: green}
}
@keyframes animateInclude-leave {
    from {opacity: 1;}
    to {opacity: 0; color: black}
}
@-webkit-keyframes animateInclude-leave {
    from {opacity: 1;}
    to {opacity: 0; color: black}
}
```

为了应用动画，需要做的就是在我们的类中添加动画CSS样式：

```
.animateInclude.ng-enter {
    -webkit-animation: 2s animateInclude-enter;
    animation: 2s animateInclude-enter;
}
.animateInclude.ng-leave {
    -webkit-animation: 2s animateInclude-leave;
    animation: 2s animateInclude-leave;
}
```

3. JavaScript动画

当用JavaScript做动画时，需要在动画的描述对象上定义`enter`和`leave`属性。

```
angular.module('myApp', ['ngAnimate'])
    .animation('.animateInclude', function() {
        return {
            enter: function(element, done) {
                // 显示如何用jQuery实现动画的例子
                // 注意，这需要在HTML中包含jQuery
                $(element).css({
                    opacity: 0
                });
                $(element).animate({
                    opacity: 1
                }, done);
            },
            leave: function(element, done) {
                done();
            }
        }
});
```

22.9.4 ngSwitch动画

`ngSwitch`指令触发这些事件：

动　　作	事件名称
新的视图内容准备好了	enter
正在离开，已有的内容将被隐藏	leave

ngSwitch指令类似于前面的例子。对于这些例子，我们用下面使用了ng-switch指令的HTML来运行：

```
<div ng-init="template='home'"
    ng-controller="HomeController">
    <button ng-click="template='home'">Home</button>
    <button ng-click="template='second'">Second</button>
    <button ng-click="template='third'">Third</button>
    <div ng-switch="template">
        <div class="animateSwitch"
            ng-switch-when="home">
            <h1>Home</h1>
        </div>
        <div class="animateSwitch"
            ng-switch-when="second">
            <h1>Second</h1>
        </div>
        <div class="animateSwitch"
            ng-switch-when="third">
            <h1>Home</h1>
        </div>
    </div>
</div>
```


1. CSS3过渡

要让ngSwitch列表中的元素动起来，我们需要确认添加了展现元素初始状态的CSS样式类，以及为enter和edit状态定义最终状态的类：

```
.animateSwitch.ng-enter,
.animateSwitch.ng-leave {
    transition: 2s linear all;
    -webkit-transition: 2s linear all;
}
```

至此，可以简单地在动画中定义初始和最终阶段的CSS属性。这里，我们把元素从绿色的文字淡入，在进入动画的最终阶段把文字变成黑色。在离开（元素移除）动画中，我们把属性反转：

```
.animateSwitch.ng-enter {
    opacity: 0;
    color: green;
}
.animateSwitch.ng-enter.ng-enter-active {
    opacity: 1;
    color: black;
}
.animateSwitch.ng-leave {}
.animateSwitch.ng-leave.ng-leave-active {
    opacity: 0;
}
```

2. CSS3动画

首先，添加为动画定义的@keyframe：

```
@keyframes animateSwitch-enter {
    from {opacity:0;}
    to {opacity:1; color: green}
```

```
}
@-webkit-keyframes animateSwitch-enter {
    from {opacity:0;}
    to {opacity:1; color: green}
}
@keyframes animateSwitch-leave {
    from {opacity: 1;}
    to {opacity: 0; color: black}
}
@-webkit-keyframes animateSwitch-leave {
    from {opacity: 1;}
    to {opacity: 0; color: black}
}
```

为了应用动画，需要做的就是在我们的类中添加动画CSS样式：

```
.animateSwitch.ng-enter {
    -webkit-animation: 2s animateSwitch-enter;
    animation: 2s animateSwitch-enter;
}
.animateSwitch.ng-leave {
    -webkit-animation: 2s animateSwitch-leave;
    animation: 2s animateSwitch-leave;
}
```

3. JavaScript动画

当用JavaScript做动画时，需要在动画的描述对象上定义`enter`和`leave`属性。

```
angular.module('myApp', ['ngAnimate'])
    .animation('.animateSwitch', function() {
         return {
            enter: function(element, done) {
                // 显示如何用jQuery实现动画的例子
                // 注意，这需要在HTML中包含jQuery
                $(element).css({
                    opacity: 0
                });
                $(element).animate({
                    opacity: 1
                }, done);
            },
            leave: function(element, done) {
                done();
             }
        }
});
```

22.9.5　`ngIf`动画

`ngIf`指令触发这些事件：

动　　作	事件名称
`ngIf`的内容变更了，新DOM被插入之后触发	`enter`
在`ngIf`的内容被移除之前触发	`leave`

对于后面的`ngIf`示例，我们用这段HTML来运行：

```
<div ng-init="show=false"
    ng-controller="HomeController">
    <button ng-click="show=!show">Show</button>
    <div ng-if="show" class="animateNgIf">
        <h2>Show me</h2>
    </div>
</div>
```

1. CSS3过渡

要让ngIf中的元素动起来，我们需要确认添加了展现元素初始状态的CSS样式类，以及为enter和edit状态定义最终状态的类：

```
.animateNgIf.ng-enter,
.animateNgIf.ng-leave {
    transition: 2s linear all;
    -webkit-transition: 2s linear all;
}
```

至此，可以简单地在动画中定义初始和最终阶段的CSS属性。这里，我们把元素从绿色的文字淡入，在进入动画的最终阶段把文字变成黑色。在离开（元素移除）动画中，我们把属性反转：

```
.animateNgIf.ng-enter {
    opacity: 0;
    color: green;
}
.animateNgIf.ng-enter.ng-enter-active {
   opacity: 1;
   color: black;
}
.animateNgIf.ng-leave {}
.animateNgIf.ng-leave.ng-leave-active {
    opacity: 0;
}
```

2. CSS3动画

首先，添加为动画定义的@keyframe：

```
@keyframes animateNgIf-enter {
    from {opacity:0;}
    to {opacity:1;}
}
@-webkit-keyframes animateNgIf-enter {
    from {opacity:0;}
    to {opacity:1;}
}
@keyframes animateNgIf-leave {
    from {opacity: 1;}
    to {opacity: 0;}
}
@-webkit-keyframes animateNgIf-leave {
    from {opacity: 1;}
    to {opacity: 0;}
}
```

为了应用动画，需要做的就是在我们的类中添加动画CSS样式：

```
.animateNgIf.ng-enter {
    -webkit-animation: 2s animateNgIf-enter;
```

```
    animation: 2s animateNgIf-enter;
}
.animateNgIf.ng-leave {
    -webkit-animation: 2s animateNgIf-leave;
    animation: 2s animateNgIf-leave;
}
```

3. JavaScript动画

当用JavaScript做动画时，需要在动画的描述对象上定义enter和leave属性。

```
angular.module('myApp', ['ngAnimate'])
    .animation('.animateNgIf', function() {
        return {
            enter: function(element, done) {
                // 显示如何用jQuery实现动画的例子
                // 注意，这需要在HTML中包含jQuery
                $(element).css({
                    opacity: 0
                });
                $(element).animate({
                    opacity: 1
                }, done);
            },
            leave: function(element, done) {
                done();
            }
        }
});
```

22.9.6　ngClass动画

当视图中的样式类发生变化时，是可以基于行为去产生动画的。当一个CSS类变更时（比如在ngShow和ngHide指令中），$animate会通知和触发动画，不管是增加了新类，还是移除了旧类。

不同于使用进入动画的命名约定，我们为ngClass使用一个新的CSS约定，依次为新类加后缀，变为[CLASSNAME]-add和[CLASSNAME]-remove。

类似于上面的进入事件，ngAnimate会在合适的时间为具体的事件添加[CLASSNAME]-add-active和[CLASSNAME]-remove-active。

当我们在这些样式类上做动画时，动画先触发，然后最终的类在动画结束时才被添加。当一个类被移除时，它直到动画结束之前都还在元素上。

ngClass指令触发这些事件：

动　　作	事件名称
ngClass求值为真之后，类被添加之前	add
类被移除之前触发	remove

对于后面的ngClass示例，我们用这段HTML来运行：

```
<div ng-init="grow=false"
    ng-controller="HomeController">
    <button ng-click="grow=!grow">Grow</button>
    <div ng-class="{grown:grow}"
```

```
        class="animateMe">
    <h2>Grow me</h2>
    </div>
</div>
```

1. CSS3过渡

要让ngClass中的元素动起来，我们需要确认添加了展现元素初始状态的CSS样式类，以及为enter和edit状态定义最终状态的类：


```
.animateMe.grown-add,
.animateMe.grown-remove {
    transition: 2s linear all;
    -webkit-transition: 2s linear all;
}
```

至此，可以简单地在动画中定义初始和最终阶段的CSS属性。

```
.grown {font-size: 50px;}
.animateMe.grown-add {
    font-size: 16px;
}
.animateMe.grown-add.grown-add-active {
    font-size: 50px;
}
.animateMe.grown-remove {}
.animateMe.grown-remove.grown-remove-active {
   font-size: 16px;
}
```

2. CSS3动画

首先，添加为动画定义的@keyframe。

```
@keyframes animateMe-add {
    from {font-size: 16px;}
    to {font-size: 50px;}
}
@-webkit-keyframes animateMe-add {
    from {font-size: 16px;}
    to {font-size: 50px;}
}
@keyframes animateMe-remove {
    to {font-size: 50px;}
    from {font-size: 16px;}
}
@-webkit-keyframes animateMe-remove {
    to {font-size: 50px;}
    from {font-size: 16px;}
}
```

为了应用动画，需要做的就是在我们的类中添加动画CSS样式：

```
.animateMe.grown-add {
    -webkit-animation: 2s animateMe-add;
    animation: 2s animateMe-add;
}
.animateMe.grown-remove {
    -webkit-animation: 2s animateMe-remove;
    animation: 2s animateMe-remove;
}
```

3. JavaScript动画

当用JavaScript做动画时，需要在动画的描述对象上定义addClass和removeClass属性。

```
angular.module('myApp', ['ngAnimate'])
    .animation('.animateMe', function() {
        return {
            addClass: function(ele, clsName, done)
            {
                // 显示如何用jQuery实现动画的例子
                // 注意, 这需要在HTML中包含jQuery
                if (clsName === 'grown') {
                    $(ele).animate({
                        'font-size': '50px'
                    }, 2000, done);
                } else { done(); }
            },
            removeClass: function(ele, clsName, done)
            {
                if (clsName === 'grown') {
                    $(ele).animate({
                        'font-size': '16'
                    }, 2000, done);
                } else { done(); }
            }
        }
});
```

22.9.7　ngShow/ngHide动画

ngShow和ngHide指令在显示或者隐藏元素时，使用了.ng-hide类。可以在显示和隐藏DOM元素之间的这段时间添加动画。

当在这些样式类上做动画时，动画会先触发，它完成时，最终的.ng-hide才会被加到DOM元素上。

因为当移除ng-hide类时，ng-hide指令还在DOM元素上，所以在它完成之前，我们是看不到动画的。因此，需要告诉CSS把我们的样式类显示出来，不要折叠。

ngShow和ngHide指令触发这些事件：

动　　作	事件名称
ngClass求值为真之后，类被添加之前	add
类被移除之后	remove

对后面的ngHide例子，我们使用下面这个HTML来运行：

```
<div ng-init="show=false"
    ng-controller="HomeController">
    <button ng-click="show=!show">Show</button>
    <div ng-show="show" class="animateMe">
        <h2>Show me</h2>
    </div>
</div>
```

1. CSS3过渡

要让ngHide中的元素动起来，我们需要确认添加了展现元素初始状态的CSS样式类，以及为enter和edit状态定义最终状态的类：

```
.animateMe.ng-hide-add,
.animateMe.ng-hide-remove {
    transition: 2s linear all;
    -webkit-transition: 2s linear all;
    display: block !important;
}
```


注意CSS块中的最后一行：它告诉CSS渲染这个类，并且，对于display属性而言，没有其他备选值。没有这行的话，这个元素就不会显示了。

至此，可以简单地在动画中定义初始和最终阶段的CSS属性。

```
.animateMe.ng-hide-add {
    opacity: 1;
}
.animateMe.ng-hide-add.ng-hide-add-active
{
    opacity: 0;
}
.animateMe.ng-hide-remove {
    opacity: 0;
}
.animateMe.ng-hide-remove.ng-hide-remove-active {
    opacity: 1;
}
```

2. CSS3动画

首先，添加为动画定义的@keyframe。

```
@keyframes animateMe-add {
    from {opacity: 1;}
    to {opacity: 0;}
}
@-webkit-keyframes animateMe-add {
    from {opacity: 1;}
    to {opacity: 0;}
}
@keyframes animateMe-remove {
    from {opacity:0;}
    to {opacity:1;}
}
@-webkit-keyframes animateMe-remove {
    from {opacity:0;}
    to {opacity:1;}
}
```

为了应用动画，需要做的就是在我们的类中添加动画CSS样式：

```
.animateMe.ng-hide-add {
    -webkit-animation: 2s animateMe-add;
    animation: 2s animateMe-add;
}
.animateMe.ng-hide-remove {
```

```
    -webkit-animation: 2s animateMe-remove;
    animation: 2s animateMe-remove;
    display: block !important;
}
```

3. JavaScript动画

当用JavaScript做动画时，需要在动画的描述对象上定义addClass和removeClass属性。

```
angular.module('myApp', ['ngAnimate'])
    .animation('.animateMe', function() {
        return {
            addClass: function(ele, clsName, done)
            {
                // 显示如何用jQuery实现动画的例子
                // 注意，这需要在HTML中包含jQuery
                if (clsName === 'ng-hide') {
                    $(ele).animate({
                        'opacity': 0
                    }, 2000, done);
                } else { done(); }
            },
            removeClass: function(ele, clsName, done)
            {
                if (clsName === 'ng-hide') {
                    $(ele).css('opacity', 0);
                    // 强制移除ng-hide类
                    // 这样
                    // 我们就可以真的把动画显示出来
                    $(ele).removeClass('ng-hide');
                    $(ele).animate({
                        'opacity': 1
                    }, 2000, done);
                } else { done(); }
            }
        }
});
```

22.10　创建自定义动画

$animate服务给我们在指令中实现自定义动画提供了帮助。把$animate服务注入到我们自己的应用中之后，可以用暴露出的事件为每个事件触发$animate对象上的关联函数。

要在我们自己的指令中开始动画，需要注入$animate服务。

```
angular.module('myApp', ['ngAnimate'])
    .directive('myDirective', function($animate) {
        return {
            template: '<div class="myDirective"></div>',
            link: function(scope, ele, attrs) {
                // 在这里添加动画
                // 例如:
                $animate['addClass'](element, 'ng-hide');
            }
        }
});
```

至此，就可以把事件绑定到指令上，开始显示我们的动画了。

建立了指令之后，我们可以调用$animate函数创建一个动画，与我们的指令通信。

```
angular.module('myApp', ['ngAnimate'])
    .animation('.scrollerAnimation', function() {
        return {
            animateFun: function(element, done) {
                // 我们可以在这个函数中
                // 做任意想做的事
                // 但是需要调用done
                // 来让angular知道动画结束了
            }
        }
});
```


$animate服务暴露了一些方法，为内置指令的动画事件提供帮助。这些$animate服务暴露出来的事件是：

- enter；
- leave；
- move；
- addClass；
- removeClass。

$animate服务把这些事件以函数的方式提供，让我们能在自己的指令中处理自定义动画。

22.10.1　addClass()

addClass()方法触发了一个基于className变量的自定义动画事件，并且把className值作为CSS类添加到元素上。当在DOM元素上添加样式类时，$animate服务给这个className添加了一个叫-add的后缀来让我们建立动画。

> 如果没有CSS过渡，在CSS选择器（[className]-add）上也没有定义关键帧动画，ngAnimate就不会触发这个动画，只是会把这个样式类加上。

addClass()方法带三个参数。

- element（jQuery/jqLite元素）

正在建立动画的元素。

- className（字符串）

正在建立动画，并且添加到元素上的CSS类。

- done（函数）

当动画完成时调用的回调函数。

```
angular.module('myApp', ['ngAnimate'])
    .directive('myDirective', function($animate) {
        return {
            template: '<div class="myDirective"></div>',
            link: function(scope, ele, attrs) {
```

```
                ele.bind('click', function() {
                    $animate.addClass(ele, 'greenlight');
                });
            }
        }
});
```

调用addClass()方法会经过如下步骤：

(1) 运行所有在元素上用JavaScript定义的动画；

(2) [className]-add类被添加到元素上；

(3) $animate检查CSS样式来寻找过渡/动画的持续时间和延迟属性；

(4) [className]-add-active类被添加到元素的classList中（触发CSS动画）；

(5) $animate用定义过的持续时间等待完成；

(6) 动画结束，$animate移除两个添加的类：[className]-add和[className]-add-active；

(7) className类被添加到元素上；

(8) 触发done()回调函数（如果定义了的话）。

22.10.2　removeClass()

removeClass()方法触发了一个基于className的自定义动画事件，并且移除在className值中定义的CSS类。当从DOM元素上移除一个类的时候，$animate服务给这个className添加了一个叫-remove的后缀来让我们建立动画。

> 如果没有CSS过渡，在CSS选择器（[className]-remove）上也没有定义关键帧动画，ngAnimate就不会触发这个动画，只是会把这个样式类加上。

removeClass()方法带三个参数。

- element（jQuery/jqLite元素）

正在建立动画的元素。

- className（字符串）

正在建立动画，并且从元素上移除的CSS类。

- done（函数）

当动画完成时调用的回调函数。

```
angular.module('myApp', ['ngAnimate'])
    .directive('myDirective', function($animate) {
        return {
            template: '<div class="myDirective"></div>',
            link: function(scope, ele, attrs) {
                ele.bind('click', function() {
                    $animate.addClass(ele, 'greenlight');
                });
```

```
            }
        }
});
```

调用removeClass()动画方法会经历如下步骤：

(1) 运行所有在元素上用JavaScript定义的动画；

(2) [className]-remove类被添加到元素上；

(3) $animate检查CSS样式来寻找过渡/动画的持续时间和延迟属性；

(4) [className]-remove-active类被添加到元素的classList中（触发CSS动画）；

(5) $animate用定义过的持续时间等待完成；

(6) 动画结束，$animate移除三个添加的类：[className]、[className]-remove和[className]-remove-active；

(7) 触发done()回调函数（如果定义了的话）。

22

22.10.3 enter()

enter()方法把元素添加到它在DOM中的父元素，然后运行enter动画。动画开始之后，$animation服务会添加ng-enter和ng-enter-active类，给指令一个机会来建立动画。

enter()方法最多可以带四个参数。

- element（jQuery/jqLite元素）

正在建立动画的元素。

- parent（jQuery/jqLite元素）

这个元素的父元素，它是我们enter动画的焦点。

- after（jQuery/jqLite元素）

这个元素的兄弟元素，它将会成为enter动画的焦点。

- done（函数）

当动画完成时调用的回调函数。

```
angular.module('myApp', ['ngAnimate'])
     .directive('myDirective', function($animate) {
        return {
            template: '<div class="myDirective">' +
                '<h2>Hi</h2></div>',
            link: function(scope, ele, attrs) {
                ele.bind('click', function() {
                    $animate.enter(ele, ele.parent());
                });
            }
        }
});
```

调用enter()动画方法会经历如下步骤：

(1) 本元素被插入父元素中，或者是after元素后面；

(2) $animate运行所有在元素上用JavaScript定义的动画；

(3) .ng-enter类被添加到元素的classList中；

(4) $animate检查CSS样式来寻找过渡/动画的持续时间和延迟属性。

(5) .ng-enter-active类被添加到元素的classList中（触发动画）;

(6) $animate用定义过的持续时间等待完成；

(7) 动画结束，$animate从元素移除.ng-enter和.ng-enter-active类；

(8) 触发done()回调函数（如果定义了的话）。

22.10.4　leave()

leave()方法运行leave动画。当它结束运行时，会把元素从DOM移除。动画开始之后，它会在元素上添加.ng-leave和.ng-leave-active类。

leave()方法带两个参数。

- element（jQuery/jqLite元素）

  正在建立动画的元素。

- done（函数）

  当动画完成时调用的回调函数。

```
angular.module('myApp', ['ngAnimate'])
    .directive('myDirective', function($animate) {
        return {
            template: '<div class="myDirective">' +
                '<h2>Hi</h2></div>',
            link: function(scope, ele, attrs) {
                ele.bind('click', function() {
                    $animate.leave(ele);
                });
            }
        }
});
```

调用leave()动画方法会经历如下步骤：

(1) $animate可运行所有在元素上用JavaScript定义的动画；

(2) .ng-leave类被添加到元素的classList中；

(3) $animate检查CSS样式来寻找过渡/动画的持续时间和延迟属性；

(4) .ng-leave-active类被添加到元素的classList中（触发动画）;

(5) $animate用定义过的持续时间等待完成；

(6) 动画结束，$animate从元素移除.ng-leave和.ng-leave-active类；

(7) 元素被从DOM移除；

(8) 触发done()回调函数（如果定义了的话）。

22.10.5　move()

move()函数触发move DOM动画。在动画开始之前，$animate服务或者把元素插入父容器中，或者直接加到after元素之后，如果有的话。动画开始后，为了动画的持续，.ng-move和.ng-move-active就会被添加。

move()方法带有四个参数。

- element（jQuery/jqLite元素）

正在建立动画的元素。

- parent（jQuery/jqLite元素）

这个元素的父元素，它是我们enter动画的焦点。

- after（jQuery/jqLite元素）

这个元素的兄弟元素，它将会成为enter动画的焦点。

- done（函数）

当动画完成时调用的回调函数。

```
angular.module('myApp', ['ngAnimate'])
    .directive('myDirective', function($animate) {
        return {
            template: '<div class="myDirective">' +
                '<h2>Hi</h2></div>',
            link: function(scope, ele, attrs) {
                ele.bind('click', function() {
                    $animate.move(ele, ele.parent());
                });
            }
        }
});
```

调用move()动画方法会经历如下步骤：

(1) 元素被移到父元素中，或者在after元素之后；

(2) $animate可运行所有在元素上用JavaScript定义的动画；

(3) .ng-move类被添加到元素的classList中；

(4) $animate检查CSS样式来寻找过渡/动画的持续时间和延迟属性；

(5) .ng-move-active类被添加到元素的classList中（触发动画）；

(6) $animate用定义过的持续时间等待完成；

(7) 动画结束，$animate从元素移除.ng-move和.ng-move-active类；

(8) 触发done()回调函数（如果定义了的话）。

22.11　与第三方库集成

22.11.1　Animate.css

Animate.css库提供了一堆很酷很有趣的跨浏览器动画。它是一个了不起的库，给我们带来强大的功能，并且无需做太多工作。

很幸运，Angular社区已经提供了一个机智的方法来把Animate.css类加到我们的Angular应用中。要使用这个Animate.css的垫片，从https://github.com/yearofmoo/ngAnimate-animate.css上下载animate.css和animate.js吧。只需在HTML中引用它们即可，如下所示：

```
<!-- HTML的头部 -->
<link rel="stylesheet" type="text/css" href="css/animate.css">
<!-- HTML的主体 -->
<script type="text/javascript" src="js/vendor/animate.js"></script>
```

至此，无需把ngAnimate当作我们应用的依赖项，只要把ngAnimate-animate.css当作依赖项包含进来就可以了。这种替代方式能运行，是因为ngAnimate-animate.css模块默认就请求了ngAnimate模块。

这个转换做完之后，我们就可以简单地通过ng-class指令来引用动画类了。

例如：

```
<div class="animateMe"
    ng-class="{'dn-fade':dn_fade}">
</div>
```

对于每个动画的可能性列表，敬请查阅说明文件[①]。

22.11.2　TweenMax/TweenLite

TweenLite和TweenMax是灵活而魔幻的库，它们是建模在ActionScript动画属性上的。要使用它们，需要确认已经下载了Greensock库。

从Greensock[②]下载它，然后存放在index.html能访问到的位置。我们建议把它放在js/vendor/TweenMax.min.js这个地方，然后要确认在页面中引用了TweetMax库：

```
<script type="text/javascript" src="js/vendor/TweetMax.min.js"></script>
```

这些都建立好了之后，就可以开始了。要在我们的应用中包含Greensock动画，需要把动画设置为使用JavaScript。在这种方式下，除了要简单地用JavaScript处理一下动画，基本就没有写集成代码的必要了：

```
angular.module('myApp', ['ngAnimate'])
    .animation('scrollAside', function($window) {
        return {
            enter: function(element, done) {
                TweenMax.set(element, {
```

① https://github.com/yearofmoo/ngAnimate-animate.css/blob/master/README.md

② http://www.greensock.com/

```
                    position: 'relative'
                });
                TweenMax.to(element, 1, {
                    opacity: 0,
                    width: 0
                });
                $window.setTimeout(done, 2000);
            }
        }
});
```

第 23 章

digest循环和$apply

让我们来看看Angular在后台是如何工作的。如何只使用几行代码就得到神奇的数据绑定？最重要的是理解$digest循环是如何工作的，以及如何使用$apply()方法。

在标准的浏览器流程中，当事件被触发时（比如点击一个链接），浏览器会执行注册给该事件的回调函数。

页面加载、$http请求返回响应、鼠标移动以及按钮被点击等情况都会触发事件。

当事件被触发时，JavaScript就会创建一个事件对象，并执行这个事件对象所在的监听特定事件的所有函数。然后它会运行JavaScript函数内的回调方法，这会回到浏览器中，还可能更新DOM。

同一时间不能运行两个事件。浏览器会等待前一个事件处理程序执行完成，再调用下一个事件处理程序。

在非Angular JavaScript环境中，可以给div的点击事件附加一个回调函数。无论何时，只要发现元素上的点击事件，这个回调函数就会运行：

```
var div = document.getElementById("clickDiv");
div.addEventListener("click", function(evt) {
    console.log("evt", evt);
});
```

你可以打开Chrome开发者工具，然后将上面的代码复制粘贴到任意页面中，再点击页面试试。

无论何时，只要浏览器检测到点击事件，就会调用使用addEventListener注册到文档上的函数。

当我们将Angular混入这个流程中时，它会扩展这个标准的浏览器流程，创建一个Angular上下文。这个Angular上下文指的是运行在Angular事件循环内的特定代码，该Angular事件循环通常被称作$digest循环。为了理解这个Angular上下文，需要看看在它里面到底发生了什么。而$digest循环有两个主要组成部分：

- $watch列表；
- $evalAsync列表。

23.1 $watch 列表

每当我们在视图中追踪一个事件时，会给它注册一个回调函数，然后希望在页面中触发该事

件时调用这个回调函数。回想一下第一个例子：

```
<!DOCTYPE html>
<html ng-app>
<head>
    <title>Simple app</title>
    <script
    src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.13/angular.js">
    </script>
</head>
<body>
    <input ng-model="name" type="text" placeholder="Your name">
    <h1>Hello {{ name }}</h1>
</body>
</html>
```


无论何时，只要用户更新这个输入字段，UI中的`{{ name }}`就会改变。发生这一变化是因为我们把UI中的输入字段绑定给了`$scope.name`属性。为了更新这个视图，Angular需要追踪变化。它是通过给`$watch`列表添加一个监控函数做到这一点的。

`$scope`对象上的属性只会在其被用于视图时绑定。在上述例子中，我们只给`$watch`列表添加了一个函数。

记住，对于所有绑定给同一`$scope`对象的UI元素，只会添加一个`$watch`到`$watch`列表中。

这些`$watch`列表会在`$digest`循环中通过一个叫做“脏值检查”的程序解析。

23.2　脏值检查

脏值检查是一个简单的过程，可归结为一个非常基础的概念：检查值是否发生了变化，而整个应用还没同步该变化。

> 除Angular之外，脏值检查策略通常用于许多不同的应用程序中。游戏引擎、数据库引擎以及对象关系映射程序（ORMs）都是这类系统很好的例子。

Angular应用持续跟踪当前监控的值（就是监控对象中那些稀奇古怪的东西）。Angular会遍历`$watch`列表，如果从旧值更新后的值没有发生变化，它会继续遍历监控列表。如果值发生了变化，该应用会启用新值并继续遍历`$watch`列表，如图23-1所示。

Anguar遍历完整个`$watch`列表，只要有任何值发生变化，应用将会退回到`$watch`循环中，直到检测到不再有任何变化。

为什么要再次运行这一循环？因为如果你更新了`$watch`列表中某个用于更新另一个值的值，Angular将检测不到更新，除非再次运行这个循环。

如果这个循环运行10次或者更多次，Angular应用会抛出一个异常，同时停止运行。如果Angular没有抛出这个异常，应用就可能进入无限循环，这是糟糕的结果。

> 在未来版本的Angular中，这个框架会使用原生浏览器规范`Object.observe()`，这将大大加速脏值检查过程。

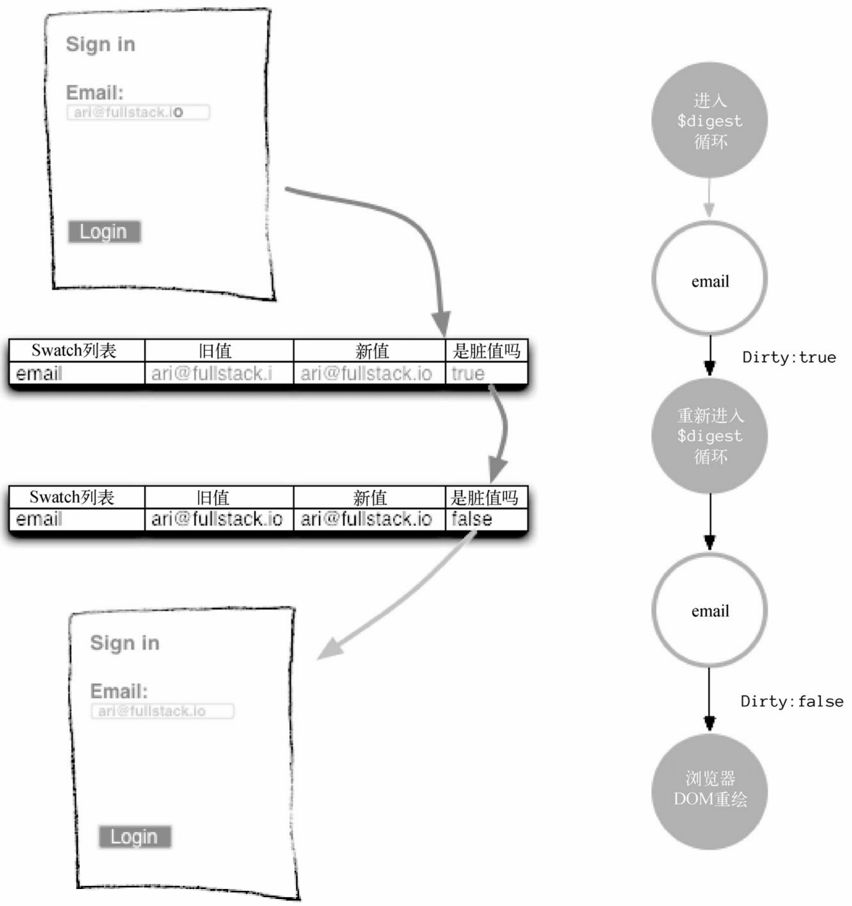


图23-1　digest循环

23.3　$watch

$scope对象上的$watch方法会给Angular事件循环内的每个$digest调用装配一个脏值检查。如果在表达式上检测到变化，Angular总是会返回$digest循环。

$watch函数本身接受两个必要参数和一个可选的参数：

- watchExpression

watchExpression可以是一个作用域对象的属性，或者是一个函数。在$digest循环中的每个$digest调用都会涉及它。

如果watchExpression是一个字符串，Angular会在$scope上下文中对它求值。如果它是一个函数，那么Angular会认为它会返回应该被监控的值。

- listener/callback

作为回调的监听器函数，它只会在watchExpression的当前值与先前值不相等（除了首次运行初始化期间）时调用。

❑ objectEquality（可选）

objectEquality是一个进行比较的布尔值，用来告诉Angular是否检查严格相等。

$watch函数会给监听器返回一个注销函数，我们可以调用这个注销函数来取消Angular对当前值的监控。

```
//...
var unregisterWatch =
    $scope.$watch('newUser.email',
        function(newVal, oldVal) {
            if (newVal === oldVal) return; // 初始化
});
// ...
// 稍后，可以通过调用这个注销函数来注销这个监控器
unregisterWatch();
```


在这个例子中，假如完成了对newUser.email的监控，那么可以通过调用它所返回的注销函数来清除这个监控器。

例如，你想要解析一个输入字段的值，然后使用空格分割全名的方式找到名字和姓氏。假定给定的视图看起来像这样：

```
<input type="text" ng-model="full_name" placeholder="Enter your full name"/>
```

永远不要在控制器中使用$watch，因为它会使控制器难以测试。这里为了解释说明姑且允许这样做，稍后我们会将这些监控移到相应的服务中。

我们在full_name属性上设置一个$watch监听器来检测值的任意变化。也就是在full_name属性上设置$watch函数。

```
angular.module("myApp")
  .controller("MyController", ['$scope', function($scope) {
    $scope.$watch('full_name', function(newVal, oldVal, scope) {
        // newVal表示在这里可以用的full_name新值
        // 而oldVal表示full_name的旧值
    });
}]);
```

在这个例子中，我们设置了一个AngularJS表达式，这会让Angular应用“监控full_name属性任何可能的变化，然后在检测到变化时运行指定的函数”。

监听函数会在初始化时被调用一次，而此时newVal和oldVal的值都为undefined（并且是相等的）。在这种情况下，如果正处在初始化阶段或者先前的值发生了变化，通常最好是检查内部的表达式。在监控函数内很容易实现这一检查，就像这样：

```
$scope.$watch('full_name',
  function(newVal, oldVal, scope) {
      if(newVal === oldVal) {
          // 只会在监控器初始化阶段运行
      } else {
          // 初始化之后发生的变化
      }
});
```

在这段代码中，$scope.$watch()函数在$scope对象上为full_name属性设置了一个监控表达式。

23.4 $watchCollection

此外，Angular还允许我们为对象的属性或者数组的元素设置浅监控，然后只要属性发生变化就触发监听器回调。

使用$watchCollection还可以检测对象或数组何时发生了变化，以便确定对象或数组中的条目是何时添加、移除或者移动的。$watchConllection的行为与$digest循环中标准的$watch的行为一样，我们甚至可以把它当作标准的$watch。

$watchCollectiion()函数接受2个参数。

- obj（字符串/函数）

这个对象就是一个要监控的对象。如果传入一个字符串，它将被当作Angular表达式求值。如果传入的是一个函数，将在当前作用域中被调用，并且会返回要监控的值。

- listener（函数）

这个回调函数会在集合发生变化时触发。类似于$watch函数，这个函数会被来自$watch的新集合触发调用，而原来的集合（先前集合的副本）以及所在的作用域也随之生效。

$watchConllection()函数也返回一个注销函数。调用这个注销函数时，也会取消集合上的$watch。

```
$scope.$watchCollection('names',
  function(newNames, oldNames, scope) {
    // names集合已经发生了变化
});
```

23.5 页面中的$digest 循环

让我们一起来看看Web页面中的$digest循环过程。首先，假设有一个（非常不可靠的）登录页面，这个页面带有一个唯一的用户名字段，允许用户使用唯一的表单验证进行登录。

我们不推荐使用这种不安全的表单验证。

```
<h2>Sign in</h2>
<input type="text" placeholder="Your name" ng-model="name" ng-minlength="3" />
<input type="submit" ng-click="login()" value="Login" />
```

这里通过ng-model指令在视图中绑定了一个name，Angular会设置一个隐式的监控器，将这个输入字段的值绑定为当前的$scope对象。

当用户输入一个字符到表单中时，Angular上下文就会生效并开始遍历$$watchers（$watch列表）。

在这个例子中，$watch列表只包含了一个唯一的元素：$scope.name。由于用户通过输入一

个字符改变了输入字段的值，这个监控函数就会在$scope.name绑定上执行。在我们退出$digest循环之前，这一行为会触发在该值（由ng-model绑定）上运行的验证和格式化操作。

由于在digest循环中值发生了变化，Angular需要再次运行这一循环以确定它没有改变作用域对象上的其他值。

> 为什么要再次运行digest循环？如果有一个名为$scope.full_name的属性由$scope.first_name + $scope.last_name组成，那么这些值的任何变化都会改变$scope.full_name，因此循环需要再次执行以确认不再有任何变化了。

因为这里只改变了$scope.name属性，并没有改变$scope对象中的其他任何属性，所以$digest循环会退出，然后浏览器会重绘DOM以刷新视图。

23

当用户在输入字段中输入他们的名字并点击提交按钮时，会引发一个略有不同的流程。

ng-click为DOM元素绑定了浏览器原生的click事件。当这个DOM元素收到点击事件时，ng-click指令会调用$scope.$apply()，同时进入$digest循环。

23.6 $evalAsync 列表

$evalAsync()方法是一种在当前作用域上调度表达式在未来某个时刻运行的方式。$digest循环运行的第二个操作是执行$$asyncQueue。可以使用$evalAsync()方法访问这个工作队列。

$digest循环期间，贯穿脏值检查生命周期的每个循环之间的队列都是空的，这意味着使用$evalAsync来调用任何函数都会发生两件事情。

- 函数会在这个方法被调用的某个时刻之后执行。
- 表达式求值之后至少会执行一次$digest循环。

$evalAsync()方法接受一个唯一参数：

- expression（字符串/函数）

这个表达式便是我们想要在当前作用域上执行的东西。如果传入一个字符串，Angular将会在当前作用域上使用$eval求值该表达式。

如果传入的是一个函数，Angular将会使用传递给这个函数的scope对象执行函数求值。

```
$scope.$evalAsync('attribute',
  function(scope) {
      scope.foo = "Executed"
});
```

使用$evalAsync时要注意的一些细节。

- 如果指令直接调用$evalAsync()，它会在Angular操作DOM之后、浏览器渲染之前运行。
- 如果控制器调用$evalAsync()，它也会在Angular操作DOM之后、浏览器渲染之前运行（永远不要使用$evalAsync()来约定事件的顺序）。

无论何时，在Angular中，只要你想要在一个行为的执行上下文外部执行另一个行为，就应该使用$evalAsync()函数。

你还可以使用它替代setTimeout()函数，但是它可能在浏览器重新渲染视图之后导致屏幕闪烁。

23.7　$apply

$apply()函数可以从Angular框架的外部让表达式在Angular上下文内部执行。例如，假设你实现了一个setTimeout()或者使用第三方库并且想让事件运行在Angular上下文内部时，就必须使用$apply()。

$apply()函数接受一个可选的参数：

- expression（字符串/函数）

这个表达式可选地接受一个字符串或函数，并且是在当前作用域内执行。

如果传入一个字符串，$apply()首先会在这个字符串上调用$eval()，以强制Angular在局部作用域上下文中使用$eval()运行字符串表达式。

如果传入一个函数，这个函数将会在所传入的函数作用域上执行。

$exceptionHandler服务会捕获和处理$eval()方法抛出的所有异常。最后，$apply()方法还会直接调用$digest循环。

```
// 使用要eval的字符串调用$apply示例
$scope.$apply('message = "Hello World"');
// 使用函数的方式并给函数传入一个作用域
$scope.apply(function(scope) {
    // 然后在函数中使用传入作用域
    scope.message = "Hello World";
});
// 使用函数时忽略作用域
$scope.$apply(function() {
    $scope.message = "Hello World";
});
// 或者通过在操作的尾部调用$apply()以强制运行$digest循环
$scope.apply();
```

简而言之，使用$scope.$apply()时可以从外部进入上下文。

如果在事件被触发时调用$apply()，就会使用Angular事件循环来运行它。如果没有调用$apply()，就不会在事件循环内执行这个函数，而它会运行在Angular上下文外部。

23.8　何时使用$apply

通常可以依赖于Angular提供的可用于视图中的任意指令来调用$apply()。所有ng-[event]指令（比如ng-click、ng-keypress）都会调用$apply()。

此外还可以依赖于一系列Angular内置的服务来调用$digest()。比如$http服务会在XHR请求完成并触发更新返回值（promise）之后调用$apply()。

无论何时我们手动处理事件，使用第三方框架（比如jQuery、Facebook API），或者调用setTimeout()，都可以使用$apply()函数让Angular返回$digest循环。

> 一般不建议在控制器中使用$apply()，因为这样会导致难以测试，而且如果不得不在控制器中使用$apply()或者$digest()，很可能让事情变得更加难以理解。

当我们将jQuery和Angular集成在一起时（这通常被视为一个肮脏的行为），就需要使用$apply()，因为Angular不会察觉到执行在Angular上下文外部的事件。例如，在使用jQuery插件时（比如datepicker），就需要使用$apply()将来自jQuery的值传递到Angular应用中。

在这里，我们构建了一个简单的指令（第10章深入探索了如何构建指令），指令中我们在元素上使用了datepicker这个jQuery插件方法。

datepicker插件暴露了一个onSelect事件，这个事件会在用户选择日期时触发。为了在Angular应用内部获取用户选择的日期，我们需要在$apply()函数内运行datepicker的回调函数。 23

> ele.datepicker()函数是由jQuery datepicker插件提供的可用于DOM元素的属性方法。要让它工作起来，需要确保在页面上引入了jQuery和jQuery datepicker插件。

> ctrl.$setViewValue()函数是在DOM元素上使用ng-model时提供的指令。更多信息请参考第5章。

```
app.directive('myDatepicker', function() {
    return function(scope, ele, attrs, ctrl) {
        $(function() {
            // 在元素上调用datepicker方法
            ele.datepicker({
                dateFormat: 'mm/dd/yy',
                onSelect: function(date) {
                    scope.$apply(function() {
                        ctrl.$setViewValue(date);
                    });
                }
            });
        });
    }
});
```

第 24 章

揭秘Angular

从本质上讲，在浏览器中加载AngularJS Web应用的方式与加载非AngularJS应用的方式一样。但是，它们的运行方式略有不同。浏览器会在构建DOM元素时加载AngularJS库（如同正常加载任意JavaScript库）。

当浏览器触发DOMContentLoaded事件时，Angular就开始工作。它首先寻找ng-app指令（更多关于ng-app指令的信息请参考第10章）。

> 加载angular.js时，如果document.readyState被设置为complete，Angular也会启动初始化。如果你想要动态连接AngularJS脚本，这个技术是有用的。

如果浏览器在DOM中找到ng-app指令，它会为我们自动启动应用。如果没有找到这个指令，Angular期望我们自己手动启动应用。

要手动启动一个AngularJS应用，可以使用Angular的bootstrap()方法。在一些罕见的情况下手动启用应用程序是有意义的。例如，想要在某个其他库的代码运行之后，或者在运行时动态创建元素时，启动AngularJS应用。

要想手动启动应用，可以像下面这样启动它：

```
var newElement = document.createElement("div");
angular.bootstrap(newElement, ['myApp']);
```

> 如果在DOM中没有找到ng-app指令，而且也没有手动启动应用，则AngularJS不会运行。忘记在页面中引入ng-app指令肯定会引发一些严重的问题。
>
> 注意，bootstrap()方法只允许我们启动angular应用一次。

如果在ng-app属性中没有指定应用程序，则Angular会加载一个不带特定模块的应用。如果指定了，Angular就会加载与这个指令关联的模块。

使用没有指定模块的ng-app：

```
<html ng-app>
</html>
```

使用带有指定模块的ng-app：

```
<html ng-app="moduleName">
</html>
```

Angular会使用ng-app指令的值配置$injector服务（第13章深入讨论了这个服务）。

一旦应用程序加载完成，$injector就会在应用程序的$rootScope旁边创建$compile服务。

$rootScope创建后，$compile服务就会接管它。它会将$rootScope与现有的DOM连接起来，然后从将ng-app指令设置为祖先的地方开始编译DOM。

24.1 视图的工作原理

当浏览器在标准的Web流程中获取HTML时，它会收到HTML代码并将它解析为一个DOM树。这个DOM树中的每个元素被称作DOM元素，这些DOM元素会构建一堆节点。然后浏览器负责绘制出这个DOM树的结构。

浏览器在提取脚本时（从<script>标签中），会暂停DOM解析并等待脚本取回（你可以修改这一行为，但是这个行为是默认的）。

当angular.js被取回时，浏览器会执行它，同时设置一个事件监听器来监听浏览器的DOMContentLoaded事件。

> DOMContentLoaded事件会在HTML文档加载完成并开始解析时触发。

检测到这个事件时，Angular会查找ng-app指令，然后创建运行需要的一系列必要的组件（即$injector、$compile服务以及$rootScope），然后开始解析DOM树。

24.1.1 编译阶段

$compile服务会遍历DOM树并搜集它找到的所有指令，然后将所有这些指令的链接函数合并为一个单一的链接函数。

然后这个链接函数会将编译好的模板链接到$rootScope中（也就是附属于ng-app所在的DOM元素的作用域）。

它会通过检查DOM属性、注释、类以及DOM元素名称的方式查找指令。

```
<span my-directive></span>
<span class="my-directive"></span>
<my-directive></my-directive>
<!-- directive: my-directive -->
```

> 更多有关指令的信息请参考第10章。

$compile服务通过遍历DOM树的方式查找有声明指令的DOM元素。当碰到带有一个或多个指令的DOM元素时，它会排序这些指令（基于指令的priority优先级），然后使用$injector服务查找和收集指令的compile函数并执行它。

指令中的compile函数会在适当的时候处理所有DOM转换或者内联模板，如同创建模板的副本。

```
// 返回一个链接函数
var linkFunction = $compile(appElement);
// 调用链接函数，将$rootScope附加给DOM元素
linkFunction($rootScope);
```

每个节点的编译方法运行之后，$compile服务就会调用链接函数。这个链接函数为绑定了封闭作用域的指令设置监控。这一行为会创建实时视图。

最后，在$compile服务完成后，AngularJS运行时就准备好了。

24.1.2 运行时

在标准的浏览器流程中，事件循环会等待事件执行（比如鼠标移动、点击、按键等）。当这些事件发生时，它们会被放到浏览器的事件队列中。如果有函数处理程序对事件作出响应，浏览器就会将event对象作为参数来调用这些事件处理程序。

```
ele.addEventListener('click', function(event) {});
```

Angular中对事件循环做了一点增强，并且Angular还提供了自己的事件循环。指令自身会注册事件监听器，因此当事件被触发时，指令函数就会运行在AngularJS的$digest循环中。

> Angular的事件循环被称作$digest循环。这个$digest循环由两个小型的循环组成，分别是evalAsync循环和$watch列表。

当事件被触发时，事件处理程序就会在指令的上下文中进行调用，也就是AngularJS的上下文中。从功能上讲，AngularJS会在包含作用域的$apply()方法内调用指令。Angular是在$rootScope上启动$digest循环时开始整个过程的，并且还会传播到所有子作用域中。

Angular进入$digest循环时，会等待$evalAsync队列清空，然后再将回调执行上下文交还给浏览器。这个$evalAsync用于在浏览器进行渲染之前，调度需要运行在当前桢栈（stack frame）之外的所有任务。

此外，$digest循环还会等待$watch表达式列表，它是一个可能在上一次迭代过程中被改变的潜在的表达式数组。如果检测到变化，就调用$watch函数，然后再次查看$watch列表以确保没有东西被改变。

> 注意，对于$watch列表中检测到的任何变化，AngularJS都会再次查看这个列表以确保没有东西被改变。

一旦$digest循环稳定下来，并且检测到没有潜在的变化了，执行过程就会离开Angular上下文并且通常会回到浏览器中，DOM将会被渲染到这里。

整个流程在每个浏览器事件之间都会发生，这也是Angular如此强大的原因。它还可以将来自浏览器的事件注入到AngularJS流程中。

第 25 章

AngularJS精华扩展

25

最受欢迎、支持最好的AngularJS插件之一是AngularUI框架。

25.1 AngularUI

AngularJS自带了很多方便的特性，你可以用它来构建富有表现力的AngularJS应用而无需依赖额外的库。然而非常活跃的AngularJS社区还构建了很多可以用来最大限度地提升应用能力的优秀的库。

本章，我们介绍AngularUI[1]库所提供的几种不同的常用组件。

目前AngularUI库已经被分割为一系列模块，所以你可以选择感兴趣的组件，而不必使用整个套件。

在我们介绍这些不同的组件时，首先需要确保安装了我们要用的每个组件。

25.2 安装

对于每个组件，你可以下载单独的JavaScript库，然后将它放置在应用程序路径中；你也可以使用Bower[2]来完成安装工作。在ng-newsletter.com[3]上我们建议使用Bower。本章我们将只会使用Bower安装每个模块。

25.3 `ui-router`

AngularUI库提供的最有用的库之一便是`ui-router`。它是一个路由框架，允许你通过状态机组织接口，而不是简单的URL路由。

25.3.1 安装

要安装`ui-router`库，你可以下载发布版本[4]的文件或者使用Bower安装。

① http://angular-ui.github.io/
② http://angular-ui.github.io/
③ http://ng-newsletter.com/
④ http://angular-ui.github.io/ui-router/release/angular-ui-router.js

要确保你已经全局安装了Bower：

```
$ npm install bower -g
```

然后你就可以使用Bower安装angular-ui库了：

```
$ bower install angular-ui-router --save
```

你还要确保在视图中链接这个库：

```
<scripttype="text/javascript"
    src="app/bower_components/angular-ui-router/release/angular-ui-router.js"></script>
```

同时还需要将ui.router作为依赖注入到你的应用中：

```
angular.module('myApp', ['ui.router']);
```

现在，不同于内置的ngRoute服务，由于ui-router基于状态工作，而不是简单的url，因此你可以将它嵌套在视图中。

在处理ngRoute服务时我们不再使用ng-view，而改为使用ui-view指令。

在ui-router内处理路由和状态时，我们主要关心的是应用程序处在哪个状态以及Web应用当前处在哪个路由位置。

```
<div ng-controller="DemoController">
    <div ui-view></div>
</div>
```

和ngRoute一样，定义在任意给定状态内的模板都处在<div ui-view></div>元素内。此外，每个模板都可以包含自己的ui-view。这事实上就允许你在路由中嵌套视图。

为了定义路由，你可以使用.config方法，和常见的方式一样，但不是将路由设置在$routeProvider上，而是将状态设置在$stateProvider上。

```
.config(function($stateProvider,$urlRouterProvider) {
  $stateProvider
    .state('start', {
      url: '/start',
      templateUrl: 'partials/start.html'
    })
});
```

这一步给状态配置对象分配了一个名为start的状态。这个状态配置对象，或者说这个stateConfig也有一些与路由配置对象相似的选项，让你能够配置应用程序的状态。

1. template、templateUrl、templateProvider

在每个视图上设置模板的方式有三种。

- template：一个HTML内容字符串或者返回HTML的函数。
- templateUrl：一个模板URL路径字符串或者是返回URL路径字符串的函数。
- templateProvider：一个返回HTML内容字符串的函数。

例如：

```
$stateProvider.state('home',{
```

```
    template: '<h1>Hello {{ name }}</h1>'
});
```

2. controller

和ngRoute一样，你可以给已经注册好的控制器关联一个URL（使用字符串），也可以创建一个控制器函数作为状态控制器。

如果没有定义模板（使用上述方式之一），就不会创建这个控制器。

3. resolve

我们还可以使用resolve功能解析要注入到控制器中的依赖列表。在ngRoute中，resolve选项允许你在路由被真实渲染之前解析promise。在angular-route内，对于可以如何使用这个选项更自由。

这个resolve选项就是一个对象，其中键就是要注入到控制器中的依赖名称，而其值就是待解析的factories。

如果传入一个字符串，angular-route会尝试匹配一个现有的已注册的服务。如果传入一个函数，则注入这个函数，而函数的返回值就是依赖。如果这个函数返回一个promise，它会在控制器被实例化之前解析，同时其值（就像ngRoute）会注入到控制器中。

25

```
$stateProvider.state('home', {
    resolve: {
        // 当结果不是promise时立即返回
        person: function() {
            return {
                name: "Ari",
                email: "ari@fullstack.io"
            }
        },
        // 这个函数返回一个promise，它会在控制器实例化之前解析
        currentDetails: function($http) {
            return $http({
                method: 'JSONP',
                url: '/current_details'
            });
        },
        // 还可以在另一个解析中使用上面返回的promise
        facebookId: function($http, currentDetails) {
            $http({
                method: 'GET',
                url: 'http://facebook.com/api/current_user',
                params: {
                    email: currentDetails.data.emails[0]
                }
            });
        }
    },
    controller: function($scope. person, currentDetails, facebookId) {
        $scope.person = person;
    }
});
```

4. url

url选项可以给应用程序的状态分配一个唯一的URL。这个url选项提供了与深度链接同样的

功能，它通过状态导航应用，而不是简单地通过URL导航。

这个选项类似于ngRoute的URL，但你可以把它当作ngRoute的升级版，稍后可以看到更多信息。

基本路由可以像这样指定：

```
$stateProvider
  .state('inbox', {
    url: '/inbox',
    template: '<h1>Welcome to your inbox</h1>'
});
```

当用户导航到/inbox时，应用会转换到inbox状态，然后使用模板内容（<h1>Welcome to your inbox</h1>）填充主要的ui-view指令。

URL可以接受一系列不同的选项，它还可以在url中设置基本的参数，就像在ngRoute中一样：

```
$stateProvider
  .state('inbox', {
    url: '/inbox/:inboxId',
    template: '<h1>Welcome to your inbox</h1>',
    controller: function($scope, $stateParams) {
        $scope.inboxId = $stateParams.inboxId;
    }
});
```

应用会捕获作为URL第二个组成部分的:inboxId。例如，如果用户转换到/inbox/1，$stateParams.inboxId就会变成1（因为$stateParams为{inboxId: 1}）。

如果你喜欢，还可以使用不同的语法：

```
url: '/inbox/{inboxId}'
```

这里路径必须与URL精确匹配。和ngRoute不同，如果用户导航到/inbox/，上面的路径能够正常工作。但是，当导航导到/inbox时，上述示例配置中的状态不会被激活。

此外，你还可以在路径参数内使用正则表达式，因此你可以设置一个匹配路由的规则。例如：

```
// 只匹配包含6个十六进制数字的inbox ID
url: '/inbox/{inboxId: [0-9a-fA-f]{6}}',
// 或者匹配每个URL中`/inbox`后面的`inboxId`（全部捕获）
url: '/inbox/{inboxId:.*} '
```

注意，不能在路由内使用正则捕获组，因为路由解析器将无法解析这个路由。

甚至还可以在路由中指定查询参数：

```
// 匹配诸如/inbox?sort=ascending形式的路由
url: '/inbox?sort'
```

5. 嵌套路由

你可以使用url参数以插入路由的方式提供嵌套路由。这让你可以在页面或者模板内有多个ui-views。例如，你可以在上面的/inbox路由内嵌套独立的路由。

```
$stateProvider
  .state('inbox', {
    url: '/inbox/:inboxId',
    template: '<div><h1>Welcome to your inbox</h1>\
        <a ui-sref="inbox.priority">Show priority</a>\
        <div ui-view></div>\
        </div>'
    controller: function($scope, $stateParams) {
        $scope.inboxId = $stateParams.inboxId;
    }
  })
  .state('inbox.priority', {
      url: '/priority',
      template: '<h2>Your priority inbox</h2>'
  });
```

第一个路由会按预期匹配（如上所示）。现在这里有了第二个路由，也就是一个匹配父路由inbox之下的子路由（因为这里我们使用.语法时会将它指定为一个子路由）。

- /inbox/1匹配第一个状态。
- /inbox/1/priority匹配第二个状态。

使用这种语法，你可以在父路由内嵌套URL。父视图中的ui-view会解析priority收件箱。

6. params

params选项是一个参数名数组或者是一个正则表达式数组。不能将这个选项与url选项联合使用。当状态被激活时，这些参数会被填充到$stateParams服务中。

7. views

ui-router的一个强大的特性就是可以在一个状态内设置多个命名视图。在独立的视图内，你可以在独立模板中定义多个要引用的视图。

如果设置了views参数，那么状态的templateUrl、template和templateProvider就会被忽略。如果你想在路由中包含父模板，就需要创建一个包含模板的抽象状态。

比方说我们有一个视图看起来像这样：

```
<div>
    <div ui-view="filters"></div>
    <div ui-view="mailbox"></div>
    <div ui-view="priority"></div>
</div>
```

现在，你可以创建命名视图来填充每个独立的模板。每个子视图都可以包含它自己的模板、控制器和使用resolve关键字解析的数据。

```
$stateProvider
  .state('inbox', {
    views: {
        'filters': {
            template: '<h4>Filter inbox</h4>',
            controller: function($scope) {}
        },
        'mailbox': {
            template: 'partials/mailbox.html'
```

```
        },
        'priority': {
            template: '<h4>Priority inbox</h4>',
            resolve: {
                facebook: function() {
                    return FB.messages();
                }
            }
        }
    }
});
```

8. **abstract**

抽象模板永远不能直接激活，但是可以设置被激活的子节点。

你可以使用抽象模板提供一个模板包装器来包裹多个命名视图，或者传递$scope对象给子节点。你还可以使用它们来传递解析后的依赖或者自定义数据，或者在同一url下嵌套多个路由（比如，所有的路由都在/adminURL之下）。

设置抽象模板与设置常规状态一样，区别只在于设置abstract属性：

```
$stateProvider
  .state('admin', {
      abstract: true,
      url: '/admin',
      template: ;<div ui-view></div>'
  })
  .state('admin.index', {
      url: '/index',
      template: '<h3>Admin index</h3>'
  })
  .state('admin.users', {
      url: '/users',
      template: '<ul>...</ul>'
  });
```

9. **onEnter、onExit**

Angular会在用户（分别）进入或者离开视图时调用这些回调函数。对于这两个选项，你可以设置希望调用的函数。这些函数可以访问被解析的数据。

这些回调函数让你可以在新视图上或者进入另一个状态时触发某个行为。使用它们可以很好地实现一个“你确定吗？”形式的模态视图，或者在用户进入这个状态之前要求用户登录。

10. **data**

你可以附加任意数据给你的状态配置对象configObject。这个选项跟resolve属性很像，但是它的数据不会被注入到控制器中，promise也不会被解析。

当需要从父状态给子状态传递数据时，这个选项特别有用。

25.3.2　事件

和ngRoute服务一样，angular-route服务会在状态生命周期的不同阶段触发不同的事件。

在应用程序内可以通过监听$scope对象的方式附加函数给这些事件。以下所有事件都会触发在$ootScope上，因此可以在任意$scope对象上监听这些事件。

1. 状态改变事件

可以使用如下方式监听这个事件：

```
$scope.$on('$stateChangeStart',
  function(evt, toState, roParams, fromState, fromParams) {
      // 可以阻止这一状态完成
      evt.preventDefault();
  });
```

这个事件可能会以如下方式触发。

$stateChangeStart 从一个状态开始过渡到另一个状态时触发这个事件。

$stateChangeSuccess 从一个状态过渡到下一个状态完成时触发这个事件。

$stateChangeError 当过渡期间发生错误时触发这个事件。通常，模板不能被解析或者解析promise失败时会引发错误。

25

2. 视图加载事件

ui-router还在视图加载阶段提供了事件。

$viewContentLoading 视图开始加载时，DOM被渲染之前，触发这个事件。

你可以像这样监听这个事件：

```
$scope.$on('$viewContentLoading',
  function(event, viewConfig) {
      // 在这里可以访问所有视图配置属性
      // 以及一个特殊的“targetView”属性
      // viewConfig.targetView
  });
```

$viewContentLoaded 在视图加载完成以及DOM渲染之后触发这个事件。

25.3.3 $stateParams

在上面的例子中，我们一直用$stateParams从URL的参数中辨别出不同的参数选项。这个服务展示了如何根据URL的不同组成部分处理数据。

例如，如果在inbox状态中有个看起来像这样的URL：

```
url: 'inbox/:inboxId/messages/{sorted}}?from&to'
```

然后用户到达这个路由：

```
/inbox/123/messages/ascending?from=10&to=20
```

那么$stateParams对象的结果就是：

```
{inboxId: '123', sorted: 'ascending', from: 10, to: 20}
```

25.3.4　$urlRouterProvider

和ngRoute一样，你可以使用路由提供程序构建规则，规定当特定的URL被激活时会发生什么。

创建的这些状态负责在不同的URL中激活自身，因此不一定需要$urlRouterProvider来管理激活和加载状态。当你想要管理发生在状态作用域之外的行为时，它就可以派上用场了，比如重定向或者身份验证。

你可以在模块配置函数中使用$urlRouterProvider。

when()　when函数接受两个参数：想要匹配的入口路径和用于重定向的路径（或者是在路径匹配时调用的函数）。

为了设置重定向，需要给when方法设置一个字符串参数。

例如，如果想将一个空路由重定向到/inbox：

```
.config(function($urlRouterProvider) {
    $urlRouterProvider.when('', '/inbox');
});
```

如果传入一个函数，它会在路径匹配时调用。这个处理程序可能返回以下三个值中的一个。

- **falsy**　这个值告诉$urlRouter该规则不匹配，同时它应该尝试找到一个不同的状态来匹配。如果想要确保用户可以正确地访问一个URL，它将很有帮助。
- **字符串**　$urlRouter会把这个字符串值当作重定向的URL。
- **truthy or undefined**　这个值让$urlRouter知道已经处理了URL。

otherwise()　和ngRoute中的otherwise()方法一样，这个otherwise()方法在没有其他路由匹配时发起重定向。这个方法是创建默认URL的一种很好的方式。

otherwise()方法接受一个参数：一个字符串或者函数。

如果传入一个字符串，任何无效或者不匹配的路由都会重定向到字符串指定的URL。

如果传入一个函数，它会在没有其他路由匹配时被调用，同时负责处理返回结果。

```
.config(function() {
    $urlRouterProvider.otherwise('/');
    // 或者
    $urlRouterProvider.otherwise(function($injector, $location) {
        $location.path('/');
    });
});
```

rule()　如果想要绕过所有的URL匹配，或者想要在操作其他路由之前对路由做一些操作，可以使用rule()函数。

使用rule()函数时必须返回一个有效路径字符串。

```
.config(function($urlRouterProvider) {
    $rulRouterProvider.rule(function($injector, $location) {
        return '/index';
    });
});
```

25.3.5 创建一个导航程序

为什么要使用比内置的ngRoute更强大的新路由方式？

当我们想要为用户创建一个注册向导的时候，就需要使用ui-router了，这是一个非常合适的应用场景。

我们将使用ui-router创建一个快速注册服务，它包含一个控制器，用于处理注册任务。

首先，需要创建应用视图：

```
<div ng-controller="WizardSignupController">
    <h2>Signup wizard</h2>
    <div ui-view></div>
</div>
```

在这个视图内，我们嵌入了注册视图。接下来，在这个注册向导中还需要有三个阶段。

- **start**：在这个阶段，我们获取用户名并向其介绍注册向导。
- **email**：在这里，接受用户的邮件。
- **finish**：此时，用户完成注册过程，我们要向其展示一个完整的页面。

在真实的应用中，finish阶段应该将注册资料发送给服务器，同时进行真实的注册操作。在这里，由于没有后端，因此暂时只显示这个视图。

这个注册程序依赖于wizardapp.controllers模块，我们将在其中编写包含控制器：WizardSignupController。

```
angular.module('wizardApp', [
    'ui.router',
    'wizardapp.controllers'
]);
```

WizardSignupController简单地提供了$scope.user对象，在注册过程以及注册行为中，我们都会使用这个对象。

```
angular.module('wizardapp.controllers', [])
  .controller('WizardSignupController',
      function($scope, $state) {
        $scope.user = {};
        $scope.signup = function() {}
  });
```

向导程序逻辑覆盖了大部分工作。你可以将这些逻辑设置到应用的config()函数中：

```
angular.module('wizardApp', [
    'ui.router', 'wizardapp.controllers'
])
.config(function($stateProvider, $urlRouterProvider) {
    $stateProvider
        .state('start', {
            url: '/step_1',
            /templateUrl: 'partials/wizard/step_1.html'
        })
        .state('email', {
            url: '/step_2',
            templateUrl: 'partials/wizard/step_2.html'
```

```
        })
        .state('finish', {
            url: '/finish',
            templateUrl: 'partials/wizard/step_3.html'
        });
});
```

设置这些选项之后，基本流程就全部完成了。现在，如果用户导航到路由/step_1，他们将被定向到流程的起点。尽管整个流程也可以都发生在根URL上（即/step_1），但你可能更希望将它们放在子路由中（例如/wizard/step_1）。

为此，只需要设置一个包装其他步骤的abstract状态就可以了。

```
.config(function($stateProvider, $urlRouterProvider) {
    $stateProvider
        .state('wizard', {
            abstract: true,
            url: '/wizard',
            template: '<div><div ui-view></div></div>'
        })
        .state('wizard.start', {
            url: '/step_1',
            templateUrl: 'partials/wizard/step_1.html'
        })
        .state('wizard.email', {
            url: '/step_2',
            templateUrl: 'partials/wizard/step_2.html'
        })
        .state('wizard.finish', {
            url: '/finish',
            templateUrl: 'partials/wizard/step_3.html'
        });
});
```

现在，这些路由不再定义在顶级路由中了，你可以将它们（子路由）安全地嵌套在/wizard URL内。

此外，我们还想在注册程序的尾部附加一个功能：在父控制器WizardSignupController上调用signup函数。我们只需在向导程序的尾部设置一个控制器来调用$scope上的函数就行了。由于整个向导程序都封装在WizardSignupController中，这就表示可以正常使用作用域嵌套作用域属性。

```
.state('wizard.finish',{
    url: '/finish',
    templateUrl: 'partials/wizard/step_3.html',
    controller: function($scope) {
        $scope.signup();
    }
});
```

25.4　ui-utils

UI工具库是一个功能强大的实用工具包，它提供了大量可用于你的项目中的自定义扩展，而你无需重新造轮子。

下面展示了ui-utils库所提供的一些值得注意的特性。

25.4.1 安装

```
$ bower install --save angular-ui-utils
```

我们需要确保在HTML模板中引入了这个库。ui-utils库的每个组件都是作为独立模块构建的，因此需要单独引入每个组件。

25.4.2 mask

当想要接受一个信用卡或者电话号码时（或者是其他任何特殊格式的信息），你可以提供一个整洁的UI来告诉用户提供干净的信息。

你需要确保在HTML中引入mask.js库：

```
<script type="text/javascript"
src="app/bower_components/angular-ui-utils/modules/mask/mask.js"></script>
```

然后将ui-mask作为依赖设置给应用：

```
angular.module('myApp',['ui.mask'])
```

现在，可以使用ui-mask指令来指定输入遮罩了。ui-mask指令接受下列形式的格式字符串：

- A ——任意字母；
- 9 ——任意数字；
- * ——任意字母数字字符。

例如，在一个input中格式化一个信用卡号码，ui-mask指令的设置看起来可能像这样：

```
<input name="ccnum" ui-mask="9999999999999999" ng-model="user.cc" placeholder="Credit card
number" />
```

> 除非所有验证都满足了，否则Angular视输入为无效，而ui-mask与此类似。
>
> 注意，上面这个input只支持输入遮罩匹配9999-9999-9999-9999的信用卡。稍微做点工作就可以支持其他类型的卡了。

同样，你可以使用字符或者任意字母数字字符格式化一个输入字段。

25.4.3 ui-event

和其他模块一样，需要在HTML中引入event.js库：

```
<script type="text/javascript"
src="app/bower_components/angular-ui-utils/modules/event/event.js"></script>
```

然后还需要将ui.event作为应用的依赖引入：

```
angular.module('myApp',['ui.event'])
```

当想要处理AngularJS自身不支持的事件时，ui-event模块极其好用。例如，如果希望用户双击某个元素或者处理一个blur事件，就必须编写一个包装函数来包装原生浏览器事件double

click。而ui-event模块就是一个简单的原生事件包装器，因此你可以使用它来响应任意元素上由浏览器触发的事件。

例如，你想在用户双击另一个图像之后显示一个图像。只需设置ui-event指令为一个由事件名称和该元素捕获到对应事件时要采取的行为组成的键–值对即可。

比如，在HTML中，你可以在控制器中设置一个双击事件dbclick调用showImage()函数：

```
<imgsrc="/images/ui/ginger.png" ui-event="{dblclick:'showImage()'}" />
```

在控制器中，可以在作用域对象上像编写标准的方法一样编写对应的方法：

```
.controller('DemoController', function($scope) {
    $scope.showImage = function() {
        $scope.shouldShowImage = !$scope.shouldShowImage;
    }
});
```

由于这个ui-event指令就是一个简单的原生浏览器事件包装器,因此你可以在任意元素上用它来模拟任意浏览器事件。

例如，如果想捕获一个元素的blur或者focus事件，也可以使用ui-event指令。

比方说想针对表单输入提供一些有用的提示。你可以在focus事件和blur事件上设置相应的行为来显示这些帮助提示。

例如，如果你有一个包含name和email输入字段的表单，可以给blur和focus事件附加一个函数来在这些输入字段上显示帮助信息。

```
<formname="form">
    <input type="text" name="name" placeholder="Your name"
ui-event="{focus: 'showNameHelp=true',
          blur: 'showNameHelp=false'}" />
    <input type="email" name="email" placeholder="Your email"
ui-event="{focus: 'showEmailHelp=true',
          blur: 'showEmailHelp=false'}" />
</form>
```

在输入字段上设置这些事件时，还可以依据用户关注的字段来显示相应的帮助信息（使用ng-show和ng-hide）。

25.4.4　ui-format

同样需要确保在HTML中引入format.js库：

```
<script type="text/javascript"
src="app/bower_components/angular-ui-utils/modules/format/format.js"></script>
```

然后设置ui.format为应用的依赖：

```
angular.module('myApp',['ui.format'])
```

format库是一个以不同方式处理字符串标记的包装器。它让你能够直接处理应用中被认为是变量的标记。

我们可以使用这个格式化库中的标记替换功能，而不是数组或者键值对形式的JavaScript对

象。例如：

```
{{ "Hello$0" | format: 'Ari' }}
```

或者，也可以在作用域中将名称绑定给变量，然后使用format库以一个干净的格式呈现它。比方说有一个看起来像这样的控制器：

```
angular.module('myApp', ['ui.format'])
  .controller('FormatController', function($scope) {
    $scope.name = 'Ari';
});
```

你还可以格式化输入字段以防范在$scope上绑定变量：

尽管这段代码并不是特别有趣（这是Angular的一项创造性的功能），但当你想要在键-值的基础上操作文本时它就变得很有趣了。

例如，你可以基于对象的键来格式化一个字符串。比方说你有一个带有name和email属性的对象：

```
.controller('FormatController', function($scope) {
    $scope.person = {
        name: 'Ari',
        email: 'ari@fullstack.io'
    };
});
```

接下来可以修改HTML，引入作为tokens的对象键，这允许你改变匹配标记来防范把键当作tokens：

```
{{ "Hello: name. Youre mail is: email" | format: person }}
```

format模块在处理翻译或者支持i18n时特别有用（更多关于翻译的信息，请参考第27章）。

第 26 章

移动应用

移动应用并不是软件开发人员的下一个领域——它们已经来了。现在已经有12亿移动Web应用用户了，而且这个数字还在不断增长（维基百科[①]）。不久之后，移动设备的数量将会超过地球上人口的数量。按照移动设备增长的速度，估计到2017年将有51亿人使用移动电话。

对于我们这些应用开发者而言，如果想要跟上时代，掌握移动开发技术很重要。对于AngularJS，在Angular团队和社区的努力下已经极大程度地支持了移动设备。

本章将通过两种不同的方式为我们的应用程序用户提供移动体验：

- 响应式Web应用；
- 基于Cordova的原生应用。

26.1 响应式 Web 应用

就Angular而言，支持移动设备最简单的方式就是使用已知和熟悉的工具——HTML和CSS来创建兼容手机的Angular应用。由于Angular本身就是基于HTML的，创造响应式设计和交互其实只需要构建一个支持不同设备的架构就够了。

26.2 交互

对于桌面应用，通过`ng-click`和熟悉的指令就能够创建交互式应用。

从Angular 1.2.0开始，我们还可以使用新的`ngTouch`模块来使用`touch`事件。由于`ngTouch`并没有内置在核心的Angular库中，因此需要先安装它。

26.2.1 安装

可以使用好几种方式安装`ngTouch`。而安装`ngTouch`模块最简单的方式就是从angular.js[②]网站下载它的源码。

找到下载部分的`extras`，然后可以下载并将`ng-touch.js`文件存储在应用可访问的位置。

或者，也可以使用Bower安装`angular-touch`：

```
$ bower install angular-touch --save
```

① http://en.wikipedia.org/wiki/List_of_countries_by_number_of_mobile_phones_in_use

② http://angularjs.org/

无论使用哪种方式，都需要在你的index.html中以脚本的方式引入这个库：

```
<script src="/bower_components/angular-touch/angular-touch.js"></script>
```

最后，还需要以依赖的方式在你的应用中引入ngTouch：

```
angular.module('myApp', ['ngTouch']);
```

现在，就可以开始使用ngTouch库了。

26.2.2 ngTouch

处理点击事件时，移动设备浏览器与桌面浏览器的工作方式略有不同。移动设备首先会检测到一个tap事件，然后等待300毫秒去检测其他点击（tap）事件（比如，等待发现用户是否在双击设备）。在这个延时之后，浏览器才会触发点击事件。

这一延迟可能让人感觉到应用反应非常迟钝。除了检测click事件，你也可以检测touch事件。

ngTouch库通过ng-click指令为我们完美地处理了这一功能，同时它还负责调用正确的点击事件。也就是说会调用所谓的快速点击事件。

在快速点击被调用之后，才会调用浏览器的延时点击，这会触发一个双击行为。ngTouch负责在ng-click事件上移除这个浏览器延时。

26

在移动设备上的浏览器中使用ngClick指令的方式，与在桌面浏览器中的方式完全相同：

```
<button ng-click="save()">Save</button>
```

ngTouch还引入了两个新指令：swipe指令。swipe指令允许我们捕获用户滑屏行为，不论是从左侧还是往右侧。swipe指令非常有用，它可以让用户通过滑动浏览器相册，或者导航到应用的其他部分。

ngSwipeLeft指令检测元素从右向左滑动，而ngSwipeRight指令检测元素从左向右滑动。

ngSwipe*指令有用的特性之一便是，它们既能用于基于touch事件的设备，也能用于鼠标点击和拖拽。

使用ngSwipe*指令很容易。比方说有一个邮件列表，我们希望它如同流行的手机邮件客户端MailboxApp一样，能够显示针对每封邮件的操作。

在这个元素列表上使用swipe指令时可以很容易实现这一功能。在显示邮件列表时，你可以在特定的邮件条目上启用某个方向的滑屏，来启动显示行为。

在显示邮件条目时，可以从反方向实现隐藏行为，如图26-1所示。

```
<ul>
    <li ng-repeat="mail in emails">
        <div ng-show="!mail.showActions" ng-swipe-left="mail.showActions=true">
            <div class="from">
                From: <span>{{ mail.from }}</span>
            </div>
            <div class="body">
                {{ mail.body }}
            </div>
```

```
            </div>
            <div ng-show="mail.showActions" ng-swipe-right="mail.showActions=false">
                <ul class="actions">
                    <li><button>Archive</button></li>
                    <li><button>Trash</button></li>
                </ul>
            </div>
        </li>
    </ul>
```

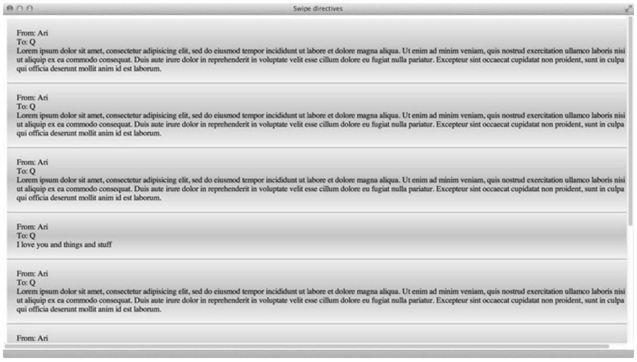


图26-1　Swipe指令示例

26.2.3　$swipe服务

可以直接使用$swipe服务实现更多自定义的基于touch的动画。这个$swipe服务简化了拖拽滑动行为的细节。

$swipe服务有一个独特的bind()方法。这个bind()方法接受一个要绑定swipe行为的元素作为参数，以及一个带有四个事件处理程序的对象。

这些事件处理程序需要一个参数对象，该对象包含一个坐标对象，就像：{x: 200, y: 300}。

这四个事件处理程序分别处理以下事件。

- start

start事件在mousedown或者touchstart事件上触发。触发这一事件之后，$swipe服务会监控touchmove和mousemove事件。而这些事件只会在移动的距离超过特定（一小段）距离时触发（为了防止意外的滑动）。

一旦超过这个特定的距离，下面两个事件之一就会发生：

- 如果垂直增量大于水平增量，浏览器将它作为滚动事件处理；
- 如果水平增量大于垂直增量，则被当作滑动行为，然后它会设置move和end事件跟踪swipe行为。

- move

实际上，在这个过程中，move事件只会在$swipe服务确定该操作是一个滑屏操作之后调用mousemove和touchmove事件。

- end

end事件在touchend或者mouseup事件之后，并且move事件被触发后触发。

- cancel

cancel事件会调用touchcancel事件或者在start事件之后开始滚动时触发。

例如，我们可以创建一个指令，在可以控制投影仪的屏幕上的幻灯片之间滑动。为了在手机设备上处理滑动操作，可以使用$swipe服务来处理如何显示UI层的自定义逻辑。

```
angular.module('myApp')
    .directive('mySlideController', ['$swipe', function($swipe) {
        return {
            restrict: 'EA',
            link: function(scope,ele,attrs,ctrl) {
                var startX, pointX;
                $swipe.bind(ele, {
                    'start': function(coords) {
                            startX = coords.x;
                            pointX = coords.y;
                    },
                    'move': function(coords) {
                            var delta = coords.x - pointX;
                            // ...
                    },
                    'end': function(coords) {
                            // ...
                    },
                    'cancel': function(coords) {
                            // ...
                    }
                });
            }
        }
    }]);
```


26.2.4 angular-gestures和多点触控手势

angular-gestures是一个Angular模块，它让我们可以处理Angular应用中的多点触控行为。它基于非常流行并经过良好测试的Hammer.js[①]库。

Hammer.js库提供了一系列常见的触屏事件：

- Tap；
- DoubleTap；
- Swipe；
- Drag；
- Pinch；
- Rotate。

angular-gestures允许我们在使用Angular指令时使用这些事件。例如，以下所有指令都是可用的：

- hmDoubleTap: 'doubletap'；
- hmDragStart: 'dragstart'；

① http://eightmedia.github.io/hammer.js/

- hmDrag: 'drag';
- hmDragUp: 'dragup';
- hmDragDown: 'dragdown';
- hmDragLeft: 'dragleft';
- hmDragRight: 'dragright';
- hmDragEnd: 'dragend';
- hmHold: 'hold';
- hmPinch: 'pinch';
- hmPinchIn: 'pinchin';
- hmPinchOut: 'pinchout';
- hmRelease: 'release';
- hmRotate: 'rotate';
- hmSwipe: 'swipe';
- hmSwipeUp: 'swipeup';
- hmSwipeDown: 'swipedown';
- hmSwipeLeft: 'swipeleft';
- hmSwipeRight: 'swiperight';
- hmTap: 'tap';
- hmTouch: 'touch';
- hmTransformStart: 'transformstart';
- hmTransform： 'transform';
- hmTransformEnd：'transformend'。

26.2.5　安装angular-gestures

为了在应用中安装angular-gestures库，需要在页面中引入getstures.js库（或者是gestures.min.js库）。

你可以直接从Github[1]下载gestures.js文件，或者也可以使用Bower进行安装。

要使用Bower安装angular-gestures，可以使用如下命令安装它：

```
$bower install --save angular-gestures
```

最后，还要设置angular-gestures作为Angular应用的依赖：

```
angular.module('myApp',['angular-gestures']);
```

26.2.6　使用angular-gestures

至此，Angular手势真的很容易使用了。手势就是Angular指令，因此在应用中使用它们与使用任何其他指令的方式一样。

① https://github.com/wzr1337/angular-gestures

比方说你想允许用户旋转，收缩和放大相册中的照片。就可以使用Hammer.js库来处理这些功能。

这里有一个例子，我们将只为元素的双击操作设置一个随机的变换。要做到这一点，需要使用hm-tap指令来设置HTML。

```
<div id="photowrapper">
    <div class="cardProps" hm-taps="tapped($event)">
        <div class="tradingcard">
            <img src="/img/ari.jpeg" alt="" />
            <span>Ari</span>
        </div>
        <div class="tradingcard">
            <img src="/img/nate.jpeg" alt="" />
            <span>Nate</span>
        </div>
    </div>
</div>
```

这里除了有一个名为hm-tap的指令之外，实际上HTML中并没有什么非常特别的东西。当有人点击图片时，angular-gestures指令会处理发生的事情。

> Hammer.js指令也可以接受Angular表达式，因此你可以在表达式内部调用函数或者执行操作（比如ng-click）以及使用Hammer.js选项。

26

在上面的例子中，调用了定义在$scope对象上的tapped()函数。我们将这个函数定义为：

```
$scope.tapped = function($event) {
    var ele = $event.target;
    var x = Math.floor(Math.random() * 200) + 1,
        y = Math.floor(Math.random() * 100) + 1,
        z = Math.flooe(Math.random() * 6) + 1,
        rot = Math.floor(Math.random() * 360) + 1;
    $(ele).css({
        'transform': "translate3d(" + x + "px," + y + "px," + z + "px)" + "rotate(" + rot +
"deg)"
    });
};
```

angular-gestures库通过提供一个叫做$event的特殊参数让我们能够访问事件对象。可以使用事件对象的target属性（$event.target）确定用户点击的是哪个元素，然后可以在元素上疯狂地使用各种优雅的特效。

26.3　Cordova 中的原生应用程序

Cordova是一个免费、开源的框架，它允许我们使用标准的Web API而不是原生代码来创建移动应用。它还允许我们使用HTML、JavaScript、CSS和AngularJS编写移动应用程序，而不是编写Objective-C或者Java（分别代表iOS和Android平台），如图26-2所示。

Cordova通过JavaScript暴露了很多可访问原生设备的API，这就允许我们运行设备特定的操作，比如获取本地位置或者使用相机功能。Cordova本身就被设计为基于插件的架构，因此你可以使用Cordova社区提供的插件，比如本地音频访问或者条形码扫描插件。



图26-2　Cordova

使用Cordova的一个好处便是可以复用Angular应用的代码来支持移动环境。当然，还有一些问题需要处理，比如性能问题和原生组件访问问题等。

安装

Cordova本身是作为一个npm包分发的，因此可以使用npm来安装它，如图26-3所示。

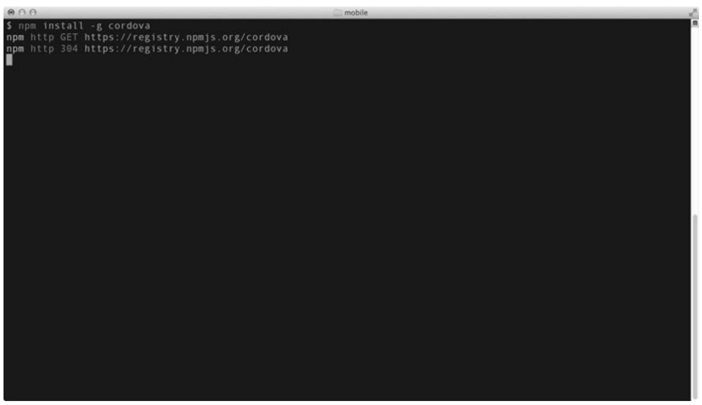


图26-3　安装Cordova

如果你还没有安装npm，首先确保安装了node。关于安装NodeJS的更多信息，请阅读第34章。

```
$ npm install -g cordova
```

Cordova程序包包含一个用于创建应用并让其可以使用Cordova的生成器。

26.4　Cordova 入门

Cordova入门很简单。首先要使用生成器创建Cordova应用的起点。让我们来看一个GapApp应用。

生成器接受3个参数。

- ❑ project-directory（必填参数）。

这是要创建应用的目录。

- ❑ package-id

项目ID（reverse-domain风格的包名）。

- ❑ name

包名称（应用程序名称）。

```
$ cordova create gapapp io.fullstack.gapapp "GapApp"
```

这行命令设置了一个叫做gapapp的目录（通过第一个参数标识），它带有一个叫做io.fullstack.gapapp的包ID，然后项目名为GapApp，如图26-4所示。

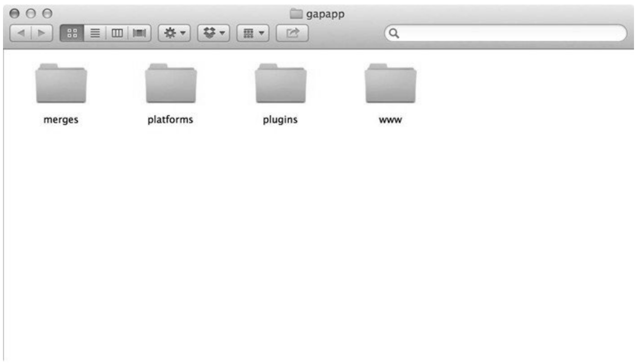


图26-4 Cordova文件结构

Cordova团队将Cordova分解成了插件，因此无需包含那些不用构建的平台；这一分解操作让开发的应用支持其他平台变得更容易。当然，也意味我们需要将感兴趣的开发平台添加到项目中。

对于上面这个项目，你可以假定其余的这些命令都运行在项目目录内：

```
$ cd gapapp/
```

比如我想构建一个iOS项目（对于其他平台，过程是一样的）。为了添加iOS平台，使用如下Cordova命令简单地将它添加到项目中即可：

```
$ cordova platform add ios
```

为了让这条命令工作起来，需要确保使用XCode安装了iOS SDK。可以在developer.apple.com[1]下载iOS SDK和XCode。

设置好之后，就可以构建基础应用了：

```
$ cordova build ios
```

现在，由于苹果公司的开发者工具有些复杂，我们不得不自己构建应用，然后让它运行在本地的iOS模拟器中。

① https://developer.apple.com/

让我们先来浏览一下应用的目录，这里会看到一个平台目录。在里面，会发现一个ios/目录，这个目录就是由上面的`platform add`命令创建的，如图26-5所示。

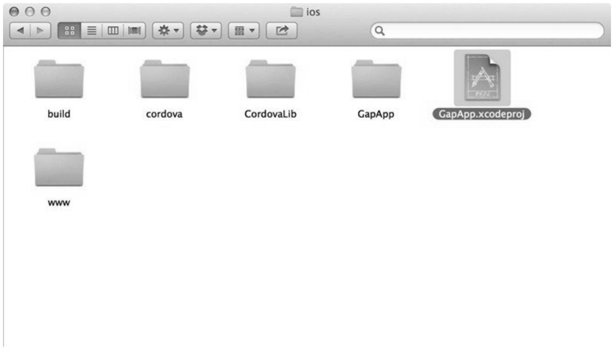


图26-5　生成的项目

在XCode中，打开使用上述命令创建的项目。确保在XCode顶部的平台标识中显示了模拟器，如图26-6所示。

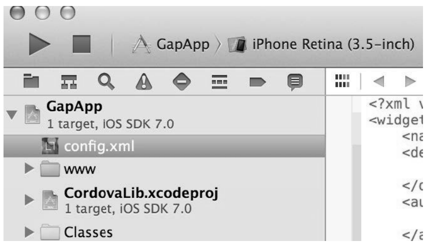


图26-6　嵌入到XCode中

点击运行。

完成这些操作后，应该能够看到这个基础的Cordova应用开始运行在模拟器中，如图26-7所示。



图26-7　标准Cordova应用

26.4.1 Cordova开发流程

Cordova源自已经被Apache基金会接受的PhoneGap项目。这个项目本身就包含了我们将用来与原生应用交互的命令行工具，从创建到部署。

26.4.2 平台

至此，我们已经创建了应用，并且添加了平台（在这个例子中是iOS）。

Cordova应用支持哪些平台，取决于我们的开发环境。在Mac上，可用的平台有：

- iOS；
- Android；
- Blackberry10；
- Firefox OS。

在Windows机器上，可以开发以下平台应用：

- Android；
- Windows Phone 7；
- Windows Phone 8；
- Windows8；
- Blackberry10；
- Firefox OS。

如果不知道哪些平台可用，可以运行`platforms`命令来检查哪些平台可用并且已经安装好了：

```
$ cordova platforms ls
```

要添加一个平台，可以使用`platform add`命令（正如上面那样）：

```
$cordova platform add android
```

要移除一个平台，可以使用`rm`或者`remove`命令：

```
$cordova platform rm blackberry10
```

26.4.3 插件

Cordova建立在令人难以置信的模块化基础之上，它希望用户使用插件系统来安装所有非核心组件。要给项目添加一个插件，使用`plugin add`命令即可：

```
$ cordova plugin add\
https://git-wip-us.apache.org/repos/asf/cordova-plugin-geolocation.git
```

然后可以使用`plugins ls`命令列出当前已安装的插件：

```
$cordova plugins ls
[ 'org.apache.cordova.geolocation' ]
```

最后，可以使用`plugin rm`命令移除插件：

```
$cordova plugins rm org.apache.cordova.geolocation
```

26.4.4　构建

默认情况下，Cordova会创建一个项目骨架，在项目目录的www/目录中存放Web视图文件。当使用Cordova构建这个项目时，它会复制这些文件，并将它们存放到平台特定的目录中。

要构建应用，可以使用另外一个Cordova命令——build命令：

```
$ cordova build
```

这里无需指定任何要构建的平台，这个命令会构建项目中列出的所有平台。

你也可以通过只构建指定平台的方式限定作用域，比如：

```
$cordova build ios
$cordova build android
```

使用build命令时要确保设置了必要的平台特定的代码，这样应用才能被编译。实际上，与调用cordova prepare和cordova compile时所做的事情一样。

26.4.5　模拟和运行

Cordova还可以运行模拟器以模拟在设备上运行应用。当然，只能在安装了模拟器并设置本地开发环境的情况下才能这么做。

假设你已经在开发环境中设置了模拟器，那么你可以让Cordova在你的模拟器中启动并安装应用。

```
$ cordova emulate ios
```

对于iOS，如果没有在机器上设置模拟器环境，你可能必须使用XCode构建项目（正如上面那样）。

此外，还可以使用run命令在特定的设备上运行你的应用程序。run命令会在设备上启动应用程序，或者在没有找到设备、没有可用设备的情况下在模拟器中启动应用。

```
$ cordova emulate ios
```

26.4.6　开发阶段

当改变了应用的某个部分时，重新编译应用以便将这部分变化反映到其中可能会比较麻烦。为了帮助开发人员加速应用Web部分的开发工作，你可以使用serve命令启用Web浏览器，为www/目录提供一个本地服务。

```
$ cordova serve ios
Static file server running at => http://0.0.0.0:8000/
CTRL + C to shutdown
```

这样就可以在Web浏览器中导航到如下URL：

```
http://localhost:8000/ios/www/index.html
```

通过使用HTTP为应用的www/目录提供服务，就可以在我们对应用作出改变时编译以及监控它。

当你对应用作出改变时，需要确保重新构建应用，如图26-8所示。

```
$cordova build ios
```



图26-8 使用Safari构建应用

26.4.7 Anguar中的Cordova服务

当你准备好Cordova应用，设备也已经连接好，一切都准备好时，Cordova会触发一个叫做deviceready的浏览器事件。

在Angular中，可以在这个事件被触发之后启动应用，或者也可以在deviceready事件被触发之后使用promise处理应用的逻辑部分。

要在接收到deviceready事件之后启动应用，需要给这个事件设置一个监听器，然后手动启动应用：

```
angular.module('myApp', []);

var onDeviceReady = function() {
    angular.bootstrap(document, ['myApp']);
};
document.addEventListener('deviceready', onDeviceReady);
```

这里推荐一个替代的方法监听deviceready事件，就是在deviceready事件被触发之后使用promise的方式设置执行绑定。

在这里我们设置了一个Angular模块监听deviceready事件。也可以使用服务来监听deviceready事件，然后依赖于这个事件是否被触发来解析promise。

```
angular.module('fsCordova', [])
  .service('CordovaService', ['$document', '$q',
    function($document, $q) {
        var d = $q.defer(),
            resolved = false;
        var self = this;
        this.ready = d.promise;

        document.addEventListener('deviceready', function() {
            resolved = true;
            d.resolve(window.cordova);
        });

        // 检查一下以确保没有漏掉这个事件（以防万一）
        setTimeout(function() {
```

```
                if(!resolved) {
                    if (window.cordova) {
                        d.resolve(window.cordova);
                    }
                }
            }, 3000);
    }]);
```

现在，再将fsCordova作为依赖，设置给应用程序模块：

```
angular.module('myApp', ['fsCordova'])
// ...
```

可以使用这个CordovaService来确定Cordova是否准备好了，事实上这里已经准备好了，并且我们还可以依赖于这个服务是否准备就绪来设置逻辑：

```
angular.module('myApp', ['fsCordova']).controller('MyController', function($scope,
CordovaService) {
    CordovaService.ready.then(function() {
        // 此时Cordova已经准备好了
    });
});
```

26.5　引入 Angular

这个空的Cordova应用，目前只是一个没有价值的JavaScript应用，它只在js/index.js中隐藏和显示了JavaScript视图。

你可以以很简单的方式将Angular引入到工作流程中。因为这里要构建一个原生应用，从CDN中引入Angular并不理想；相反，应该将必要的组件直接包含到应用中。

虽然可以使用Bower处理更复杂的设置，但是就目前而言，先保持简单。

为了构建Angular应用，需要从angularjs.org[①]上下载Angular，并且应该将它存放到index.html可访问的目录中。推荐www/js/vendor/angular.js。

设置好之后，就可以开始构建Angular应用了。首先需要在www/index.html中引入这个JavaScript文件。

```
<script type="text/javascript" src="js/vendor/angular.js"></script>
```

现在，你可以替换当前Angular应用中js/index.js文件中的所有内容，然后正常开发这个应用了。

开发流程

构建应用时，我们将会使用如下流程：

- 启动本地服务器（Cordova serve [platform]）；
- 编写应用；
- 重新构建应用（Cordova build [platform]）。

① http://angularjs.org/

这个流程虽然有些麻烦，但这就是我们编写应用的方式。

如果你的应用不依赖于Cordova平台，那可以编写脱离模拟器的应用，在浏览器中运行应用。这种情况下，你就可以专心构建应用，而无需重新构建或者重新部署应用。

26.6 使用 Yeoman 构建

你可以使用Yeoman①来构建生产就绪的应用。Yeoman是一个构建脚本的集合，这是一个官方支持的Angular应用构建程序。关于Yeoman的更多信息，请参考34.7节。

先安装Yeoman、Angular生成器以及Cordova生成器：

```
$ npm install -g yo
$ npm install -g generator-angular
$ npm install -g cordova
```

为了同时使用Yeoman和Cordova，我们还需要对上述流程做一些调整。

首先要创建一个标准的Cordova应用：

```
$ cordova create gapapp io.fullstack.gapapp "GapApp"
```


这行命令会在本地目录中创建一个标准的gapapp目录，如图26-9所示。

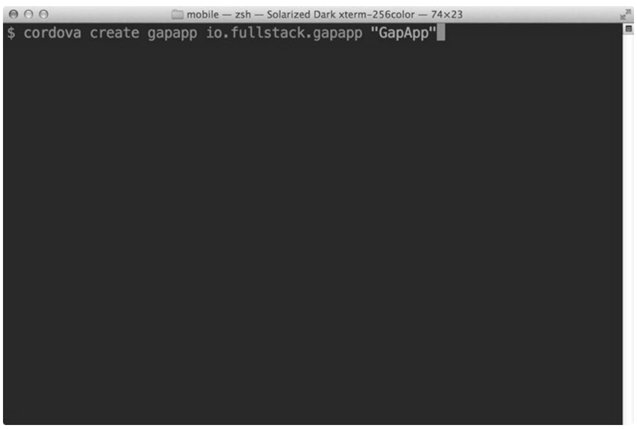


图26-9 生成应用

接下来，让我们进入到这个目录并添加平台：

```
$ cd gapapp/
$ cordova platform add ios
```

这一步会创建一个平台目录，稍后我们将会在模拟器和设备中在本地处理真实的gapapp。

要做的第一件事就是在目录中设置Yeoman应用，然后对默认配置做一些细微的改变。

```
$ yo angular
```

这回通过标准的Yeoman问题完成这一过程，同时会构建一个标准的目录。

① http://yeoman.io/

当这个过程完成时，在app/目录中就有Yeoman应用了。我们将会在这个位置进行构建移动应用的所有工作。

在这个工具链中，我们将使用如下流程来构建这个应用：

- 编写代码；
- 测试代码（Angular测试）；
- 在模拟器中运行代码（可选）；
- 运行设备测试代码（可选）。

Yeoman构建工具负责前面两个任务。后面两个任务稍后会建立。

Cordova的工作原理就是将www/目录包含到编译后的应用中，因此对www/目录中的文件做出的任何改变都将会在构建后被包装到编译好的应用程序中。

26.6.1　修改Yeoman以便使用Cordova

默认情况下Yeoman会采用一个不同的结构将应用程序构建到dist/目录中。我们将修改这个构建目录，让它应用始终构建到www/目录中。

首先，必须保存由Cordova的创建命令构建的www/config.xml文件。我们希望把它从www/中复制或者移动到app/目录中。

```
$ cp www/config.xml app/
```

我们在使用Yeoman构建应用时将这个config.xml文件复制回www/目录中。

为了改变使用Yeoman构建应用时的默认目录，在Gruntfile.js中找到Yeoman配置部分，然后将`dist:`属性的值从`dist`改变为`www`即可：

```
// ...
grunt,initConfig({
    yeoman: {
        app: require('./bower.json').appPath || 'app',
        dist: 'www' // <~ 改变这个选项为www
    },
    watch: {
        // ...
```

接下来，你需要告诉Yeoman，在它复制的文件列表中要包含config.xml文件。幸运的是，这个过程很容易：只需添加一个字符串到复制任务的文件路径中即可。

在`copy:dist`配置内，将xml扩展名添加到复制文件列表的匹配模式中就行了：

```
// ..
},
copy: {
    dist: {
        file: [{
            expand: true,
            dot: true,
            cwd: '<%= yeoman.app %>',
            dest: '<%= yeoman.dist %>',
            src: [
```

```
                '*.{ico,png,txt,xml}', // <~添加xml扩展名
                '.htaccess',
            // ...
```

这两个命令就绪后，我们就可以使用Yeoman构建应用了，它将会构建app/目录以及www/目录内的应用程序。

```
$ grunt build
```

这条命令会设置这个基础应用使用Yeoman，但是对那些我们想要用来真正构建移动应用的开发工具并不适用。

26.6.2　装配Yeoman构建

注意，在开发应用程序时，我们不能从CDN提取远程资源。默认情况下，Yeoman会设置应用从Google CDN加载脚本。因此，必须稍微修改一下index.html模板，通过将script标签包括在`usemin`构建脚本内的方式放弃从CDN加载`jquery`和`angular`脚本，就像这样：

```
<!--
    添加下面这行配置，然后在script标签之后添加一个endbuild
-->
<!--build:jsscripts/library.js-->
<script src="bower_components/jquery/jquery.js"></script>
<script src="bower_components/angular/angular.js"></script>
<!--endbuild-->
```

26

26.6.3　构建移动部分

为了使用Yeoman这个工具构建移动应用，你可以给Grunt添加一些任务定义，从而创建一个构建、测试和部署到设备的程序。

Cordova有两个不同的用于构建应用的文件（二进制文件）：一个是platforms/目录中的本地构建工具（用于真实的编译原生应用程序）和一个全局二进制文件（用于构建通过Cordova的`create`命令创建在根目录中的应用程序）。

为了正常构建应用程序，你可以从根目录运行Cordova构建命令。Cordova会复制www/目录到不同平台的适当位置。

```
$ cordova build
```

你也可以创建一个任务来做这件事，这样就可以使用Grunt进行标准构建工作。为了支持这一点，需要安装一个叫做`grunt-shell`的Grunt库。你可以使用npm安装这个库。

```
$ npm install --save-dev grunt-shell
```

接下来，还需要定义`shell`命令配置。我们将创建两个命令：一个用于在计算机的移动设备模拟器中模拟运行应用，另一个用于将应用部署到真实的设备中。

```
uglify: {
    // ...
},
shell: {
    build: {
```

```
            command: 'cordova build'
        },
        emulate: {
            command: 'cordova build'
        }
    }
    // ...
```

目前为止这两条命令非常相似，但是对于这两条命令，我们希望让Cordova首先构建应用程序。

因此我们需要使用局部的Cordova命令模拟或者运行我们的应用程序。局部的Cordova命令可以在每个平台类型对应的平台路径中找到。

例如，由于我们已经在platforms/ios/cordova/emulate中添加了iOS平台，因此在iOS目录中有一个局部的`emulate`命令。如果添加了Android平台，还可以在platforms/android/cordova/emulate这个Android平台目录内找到这个Cordova命令。

我们还希望构建一个辅助函数来帮助我们找到这些局部Cordova命令。在Gruntfile的顶部，添加如下命令即可：

```
module.exports = function(grunt) {
    var path = require('path'),
        cordova = require('cordova');

    var cordova_cmd = function(cmd) {
        var target = grunt.option('target') || "ios";
        return path.join(__dirname, "platforms", target, "cordova", cmd);
    }
}
```

现在，就可以使用`cordova_cmd()`函数找到这些局部Cordova命令了。对于这些局部命令，还可以修改前面建立的shell任务来引入这些自定义的任务。

```
shell: {
    build: {
        command: 'cordova build &&' +
            cordova_cmd('emulate')
    },
    run: {
        command: 'cordova build &&' +
            cordova_cmd("run")
    }
}
```

现在，就可以通过在shell中直接运行它们来使用这些命令进行测试了：

```
$ grunt shell:build
```

> 基于我们正在开发的平台，可能还需要安装相关依赖。例如，对于iOS平台，需要确保安装了`ios-sim`。更多信息可以参考Cordova官方文档[①]提供的平台依赖。

由于还没有将这些命令包装到其他命令中来真正地从app/目录将应用构建到www/目录中，因此它们基本上没什么用。

① http://docs.phonegap.com/en/3.1.0/index.html

Grunt让这一过程很变得容易：可以简单的注册一个新任务来运行多个Grunt任务。在这种情况下，你可以简单地将build和run命令包装到一个优先调用build的新任务中：

```
    // 这里是上面的任务配置
    }
});
// ...
grunt.registerTask('devemulate', [
    'build',
    'shell:build'
]);

grunt.registerTask('devrun', [
    'build',
    'shell:run'
]);

grunt.registerTask('server',function(target){
    // ...
```

这段代码让我们可以使用devemulate命令：这样就可以在模拟器环境或者设备中运行我们的应用。

```
$ grunt devemulate
```


> 注意，如果这个命令没有按照预期工作，通常可以使用grunt的--verbose标记揭示问题所在，比如缺少依赖。

此外还可以使用devrun任务在一个已经设置接受正在开发的应用程序的移动设备中运行这个应用程序：

```
$ grunt devrun
```

26.6.4 处理引导程序

最后，Cordova平台使用触发在DOM上的deviceready事件来表明设备已经准备就绪了。但是在Cordova应用已经准备好和Angular应用启动之前，两者之间又陷入了一个时机问题。可以通过创建一个捕获deviceready事件的服务，然后将它转变为一个在外部可用的变量来避开这个问题。这个服务非常简单：

```
angular.module('gapappApp.services')
  .factory('Cordova', function($q) {
    var d = $q.defer();
    if(window.navigator) {
        d.resolve(window.navigator);
    } else {
        document
        .addEventListener('deviceready', function(evt) {
            d.resolve(navigtor);
        });
    }
    return {
        navigator: function() {
            return d.promise;
```

```
            }
        }
    });
```

现在，当想要使用Cordova的`navigator`时，只需遵循如下语法形式，设备就绪后就能解析它了：

```
angular.module('gapappApp')
  .controller('MainController',
    function($scope, Cordova) {
        Cordova.navigator().then(function(navigator) {
            navigator.notification.vibrate();
        });
  });
```

第 27 章

本地化

27

随着世界各地Web访问量的增加，作为开发者的我们也在不断让应用国际化、本地化。当用户访问我们的应用时，他应该能够在运行时立即切换语言环境。

鉴于我们正在开发的是AngularJS客户端应用，尤其不希望用户必须刷新页面或者访问一个完全不同的URL。当然，AngularJS可以很容易地调整那些国际化读者的本机语言环境，或许通过为不同语言生成不同模板的方式为应用提供服务。

然而，这个过程可能会很麻烦，当我们想要改变应用的布局时会发生什么情况？每个独立的模板都需要重新构建和部署。而这个过程应该是很简单才对。

27.1 angular-translate

你可以使用angular-translate来替代创建新模板的方式，这个AngularJS模块为你的应用提供了i18n（国际化）服务。angular-translate要求创建一个JSON文件，它描述每种语言的翻译数据。然后它只会在必要时从服务器延迟加载特定语言的翻译数据。

angular-translate库自带了很多内置指令和过滤器，这让我们的应用国际化变得简单。我们一起来学习一下。

27.2 安装

为了使用angular-translate，需要加载这个库。可以使用几种不同的方式安装它，但是推荐使用Bower。

Bower是一个前端包管理器。它不仅能够处理JavaScript库，还可以处理HTML、CSS以及图片程序包。一个程序包就是一个简单的封装，典型的例子就是一个可公开访问的第三方代码库。

- 使用Bower

使用标准的Bower方法安装angular-translate：

```
$ bower install angular-translate
```

此外，你也可以从Github下载压缩版的angular-translate。

安装好最新稳定版本的angular-translate之后，你就可以简单地将它嵌入到你的HTML文

档中。只要确保它嵌入在Angular脚本之后，因为它依赖于angular库。

```
<script src="path/to/angular.js"></script>
<script src="path/to/angular-translate.js"></script>
```

最后一项要点是，在你的应用中必须将angular-translate声明为一个加载依赖：

```
var app = angular.module('myApp', ['pascalprecht.translate']);
```

很好！现在已经准备好使用angular-translate组件来翻译你的应用了。

27.3　教你的应用一种新语言

安装好angular-translate后，将它声明为应用的依赖，这样才可以用它来翻译应用程序的内容。

首先，需要提供翻译数据，这样应用才能真正地说一种新的语言。这一步可以通过使用最新的$translateProvider服务配置$translate服务实现。

培养应用使用一种新的语言很简单。只需在应用上使用config函数，为应用提供不同的语言翻译（比如英语、德语、希伯来语等）。首先，需要将$translateProvider注入到配置函数中，就像这样：

```
angular.module('angularTranslateApp', ['pascalprecht.translate'])
    .config(function($translateProvider) {
        // 翻译会放到这里
});
```

要添加一种语言，必须让$translateProvider找到一个*翻译表格*，这是一个JSON对象，它包含将要翻译为具体值的消息（键）。使用*翻译表格*时允许我们将翻译数据编写为一个简单的JSON文件，以便从远程加载或者在编译时设置，比如：

```
{
    'MESSAGE': 'Hello world',
}
```

在翻译表格中，键（key）表示一个翻译ID，而其值表示某种语言特定的翻译数据。现在，先给应用添加一个翻译表格。$translateProvider提供了一个叫做translations()的方法来处理它。

```
app.config(function($translateProvider) {
    $translateProvider.translations({
        HEADLINE: 'Hello there, This is my awesome app!',
        INTRO_TEXT: 'And it has i18n support!'
    });
});
```

有了这个翻译表格之后，应用就可以使用angular-translate了。由于这是在配置期间添加的翻译表格，因此在angular-translate的组件被实例化时就能访问到这个翻译表格了。

先切换到应用的模板部分。要将键绑定给视图很简单，只需添加翻译数据到视图层即可。使用translate过滤器时，甚至不必接触控制器或者服务，或者无需担心视图层：因为你可以从任何控制器或者服务中解耦翻译逻辑，并且无需接触业务逻辑代码就能让视图可替换。

从根本上说，translate过滤器的工作原理就像这样：

```
<h2>{{ 'TRANSLATION_ID' | translate }}</h2>
```

也可以使用这个translate过滤器来更新示例应用程序：

```
<h2>{{'HEADLINE' | translate }}</h2>
<p>{{'INTRO_TEXT' | translate}}</p>
```

很好！现在可以翻译视图层中的内容了，并且还避免了翻译逻辑污染控制器逻辑；然而，即使我们不使用angular-translate也能得到完全相同的结果，因为在这个示例应用中只涉及一种语言。

接下来我们一起看看angular-translate真正的能力，然后学习如何教应用多种语言。

27.4 多语言支持

前面我们已经通过translations()方法为应用添加了一个翻译表格。

正如translations()方法所设置的，$translateProvider识别了一种语言。现在，可以通过提供第二个*翻译表格*以同样的方式添加一种新的语言。

设置第一个翻译表格时，我们可以给它提供一个键（语言键）来指定我们要翻译的语言。这样可以使用不同的语言键添加不同的翻译。

更新一下应用，让它包含第二种语言：

```
app.config(function($translateProvider) {
    $translateProvider.translations('en_US', {
        HEADLINE: 'Hello there, This is my awesome app!',
        INTRO_TEXT: 'And it has i18n support!'
    });
});
```

为了给不同语言添加附加的翻译表格，比方说德语，我们只需使用不同的语言键做同样的事情就行了。

```
app.config(function($translateProvider) {
    $translateProvider.translations('en', {
        HEADLINE: 'Hello there, This is my awesome app!',
        INTRO_TEXT: 'And it has i18n support!'
    })
    .translations('de', {
        HEADLINE: 'Hey, das ist meine großartige App!',
        INTRO_TEXT: 'Und sie untersützt mehrere Sprachen!'
    });
});
```

现在，应用能识别两种不同的语言了。你可以根据需要随意添加尽可能多的语言；但是，现在这里有了两种有效的语言，那么应用如何知道该使用哪种语言呢？在你告诉它应该怎么做之前，angular-translate不会推荐任何语言。

要设置*首选语言*，你可以使用$translateProvider.preferredLanguage()方法。这个方法会告诉angular-translate哪种已注册语言是应用默认应该使用的语言。它要求使用一个语言键

值作为参数，这个参数指向某个翻译表格。

现在，我们来告诉应用应该使用英语作为默认语言：

```
app.config(function($translateProvider) {
    $translateProvider.translations('en', {
        HEADLINE: 'Hello there, This is my awesome app!',
        INTRO_TEXT: 'And it has i18n support!'
    })
    .translations('de', {
        HEADLINE: 'Hey, das ist meine großartige App!',
        INTRO_TEXT: 'Und sie untersützt mehrere Sprachen!'
    });
    $translateProvider.preferredLanguage('en');
});
```

27.5　运行时切换语言

要在运行时切换到一种新语言，必须使用angular-translate的$translate服务。它有一个use()方法，这个方法要么返回一个当前正在使用的语言对应的语言键，或者传递一个语言键作为参数，这个参数会让angular-translate使用该参数对应的语言。

为了感受一下如何在真实的应用中运用这一功能，我们可以添加两个表示译文的翻译ID，稍后添加到HTML模板中的按钮会用到这两个ID：

```
app.config(function($translateProvider) {
    $translateProvider.translations('en', {
        HEADLINE: 'Hello there, This is my awesome app!',
        INTRO_TEXT: 'And it has i18n support!',
        BUTTON_TEXT_EN: 'english',
        BUTTON_TEXT_DE: 'german'
    })
    .translations('de', {
        HEADLINE: 'Hey, das ist meine großartige App!',
        INTRO_TEXT: 'Und sie untersützt mehrere Sprachen!'
        BUTTON_TEXT_EN: 'englisch',
        BUTTON_TEXT_DE: 'deutsch'
    });
    $translateProvider.preferredLanguage('en');
});
```

接下来，我们要使用$translate服务和它的use()方法在控制器上实现一个方法以便在运行时改变语言。为此，要将$translate服务注入到应用的控制器中，然后添加一个函数给它的$scope对象：

```
app.controller('TranslateController', function($translate, $scope) {
    $scope.changeLanguage = function(langKey) {
        $translate.use(langKey);
    }
});
```

接着，我们通过为每种语言添加一个按钮的方式在HTML模板中反映这一变化。还必须在每个按钮上设置一个ng-click指令在运行时调用改变语言的函数。

```
<div ng-controller="TranslateController">
    <button ng-click="changeLanguage('de')" translate="BUTTON_TEXT_DE"></button>
    <button ng-click="changeLanguage('en')" translate="BUTTON_TEXT_EN"></button>
</div>
```

瞧瞧！现在我们就拥有一个支持多语言的应用了。

27.6 加载语言

如果我们只是设置静态语言那多没劲啊！多亏了Angular的$http服务，我们可以通过$translateProvider的registerLoader方法来动态加载语言。

首先，需要安装angular-translate-loader-url扩展来设置loader-url服务，它要求有一个后端服务器通过处理lang参数的方式返回JSON数据。如果你已经有一个处理带lang参数路由的后端程序，那么可以像这样使用Bower安装loader-url服务：

```
$ bower install angular-translate-loader-url
```

如果你更喜欢使用服务来加载静态文件，那么可以使用static-files加载器从语言路径中加载JSON文件。由于这个路由程序很简单，这里将继续使用Bower来安装这个服务：

```
bower install angular-translate-loader-static-files
```

现在，先让我们确保已经使用script标签将这个文件载入视图中了：

```
<script src="/js/angular-translate-loader-url.min.js"></script>
```

27

为了配置服务以使用这个static-files加载器，你需要让$translateProvider使用一个配置对象来启用这个加载器。这个配置对象接受两个参数：

- prefix指定文件前缀（含文件路径）；
- suffix指定文件后缀（常见的扩展名）。

这个加载器试图从如下URL路径中获取文件：[prefix]/[langKey]/[suffix]。比如，如果设置配置对象为：

```
$translateProvider.useStaticFilesLoader({
    prefix: '/languages/',
    suffix: '.json'
});
```

angular-translate会试图从/languages/en_US.json中加载en_US语言。像这样使用StaticFilesLoader时就带来了*延迟加载*的额外好处。在运行时$translate只会在需要语言文件时才加载它。

当然，应用加载时，异步加载会导致未翻译的内容闪现。可以通过将应用自带语言设置为默认语言的方式规避这一副作用。

最后还有一个很酷的特性：可以使用本地存储（local storage）功能存储语言文件。angular-translate提供了使用本地存储的能力；你可以用一个函数来启用这一功能：

```
.config(function($translateProvider) {
    $translateProvider.useLocalStorage();
});
```

至此我们已经介绍了如何使用anguar-translate的$translateProvider.translations()方法和translate过滤器为Angular应用提供i18n支持。此外还展示了如何使用$translate服务以及它的use()方法在运行时改变语言。

你可以尝试一下angular-translate。它提供了很多非常棒的特性，比如处理多元化，使用自定义加载器和通过服务设置翻译。它的文档非常出色，建议你在这里①查看它们。

那里还有很多可以直接用于你的站点的示例。同时它还是一个展示了所有可用组件和接口的API参考手册②，这些都可以用于构建了不起的带有国际化支持的应用。

27.7　angular-gettext

类似于angular-translate，angular-gettext使用一个完不同方法提供了翻译功能。但它不需要我们将想要翻译的字符串嵌入到应用中，而是抽象出特定的字符串让程序库处理它们。

Gettext是一个由GNU项目发起的国际化和本地化系统，该项目最早发布于1995年。它是一个非常流行的支持新语言的系统，因为它可将字符串打包，以便稍后翻译。

angular-gettext库以同样的方式工作，它将稍后想要翻译的字符串包装起来。

27.8　安装

为了使用angular-gettext，首先需要加载anglar-gettext库。虽然可以使用几种不同的方式安装它，但是我们推荐使用Bower。

- 使用Bower

可以像这样使用Bower安装angular-gettext：

```
$ bower install --save angular-gettext
```

此外，你也可以从Github下载压缩版的angular-gettext。

安装了最新的稳定版的angular-gettext之后，就可以简单的将它嵌入到HTML文档中。只需要确保在Angular之后嵌入，因为它依赖于核心的angular库以及jquery（因为它也是angular-gettext的依赖项）。

```
<script src="path/to/jquery.js"></script>
<script src="path/to/angular.js"></script>
<script src="path/to/angular-gettext.js"></script>
```

最后一项要点是，在你的应用中必须将angular-gettext声明为一个加载依赖：

```
var app = module('myApp', ['angular-gettext']);
```

现在就已准备好使用angular-gettext的组件来翻译你的应用了。

① http://pascalprecht.github.io/angular-translate

② http://pascalprecht.github.io/angular-translate/#/api

27.9 用法

angular-gettext库包含translate指令，这是一个简单的指令，它可以被放置在那些包含你想要翻译的字符串的DOM元素上。

```
<h1 translate>Hello!</h1>
```

<h1>的内容将使用我们稍后会定义的字符串译本自动翻译。

与普通的字符串相比，待翻译的字符串也会以同样的方式处理，angular-gettext提供的功能完全支持从应用内部插值。

```
<h1 translate>Hello {{name}}</h1>
```

这还可以支持翻译复数表示法。例如，比如你想要将apple翻译为它的复数形式apples。

```
<h1 translate>One apple</h1>
```

你可以添加两个甚至更多指令给<h1>元素，来表示当前计数以及要翻译的最终字符串。

```
<h1 translate
translate-n="count"
translate-plural="{{ count }} apples">
One apple
</h1>
```

如果translate-n表达式的结果大于1，那么gettext就使用translate-pural中的字符串；否则，它使用DOM元素<h1>的值。

27

这个附加的translate-n指令接受任意形式的Angular表达式，包括函数。关于表达式的更多信息，请查看第6章。

translate-pural就是一个简单的字符串，它会在这个指令被调用时替换DOM元素内的值。

最后，你还可以在应用内使用translate过滤器。因为有时候我们不能使用指令，例如：

```
<input type="text" placeholder="Username"/>
```

还可以使用translate过滤器来替换占位符中Username的值。

```
<input type="text" placeholder="{{'Username' | translate }}"/>
```

27.10 字符串提取

现在，我们不再提前提供需要翻译的字符串，而是从模版中提取字符串来构建翻译。我们将会生成一个.pot文件，也就是标准的gettext模板。

要提取那些要翻译的字符串，最简单的方法就是使用grunt-angular-gettext工具。

关于Grunt的更多信息，请查看34.3节。

为了使用grunt-angular-gettext这个Grunt任务，首先需要使用npm安装它：

```
$ npm install grunt-angular-gettext --save-dev
```

这个Grunt任务安装好之后，还需要在Gruntfile中加载它。使用标准的Grunt方法引用Grunt任务时，可以像这样在Gruntfile内部启用它：

```
grunt.loadNpmTasks('grunt-angular-gettext');
```

设置好之后，还需要从应用中提取要翻译的字符串。可以使用nggettext_extract任务做到。

为了设置这个任务，需要提供相应的配置信息。

实际上，在nggettext_extract任务中最重要的是files属性：

```
grunt.initConfig({
    nggettext_extract: {
        pot: {
            files: {
                'po/template.pot': ['src/view/*.html']
            }
        }
    }
});
```

此外，还可以在任务中包含一些选项，设置起始和终止标识符。如果要配置Angular使用不同的分隔符，那么可以在这个任务中设置这些选项：

```
grunt.initConfig({
    nggettext_extract: {
        pot: {
            options: {
                startDelim: '//',
                endDelim: '//'
            },
            files: {
                'po/template.pot': ['src/view/*.html']
            }
        }
    }
});
```

这样，就可以使用Grunt运行这个任务了，就像这样：

```
$ grunt nggettext_extract
```

这个任务运行完成后，将会得到一个po/template.pot文件。例如，对于这个模板：

```
<div ng-controller="HomeController">
    <h1 translate>Hello {{ user.name }}</h1>
    <h2 translate translate-n="count" translate-plural="{{ count }} books">{{ count }} books
</h2>
</div>
```

这会得到一个po/template.pot文件，它看起来像：

```
msgid ""
msgstr ""
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
```

```
#: app/index.html
msgid "{{ count }} books"
msgid_plural "{{ count }} books"
msgstr[0] ""
msgstr[1] ""
```

27.11 翻译字符串

现在我们已经有准备好的.pot文件了，可以开始翻译它了。使用开源软件的一个主要原因是可以使用很多工具为我们有效地创建翻译数据。

这里将重点关注如何使用Poedit工具，这是一个能够编辑.pot文件的开源工具。

首先，我们需要下载这个工具。可以从Poedit网站www.poedit.net[①]得到它。

安装好这个工具后，就可以打开应用程序，然后选择“文件→新建POT文件…”，如图27-1所示。

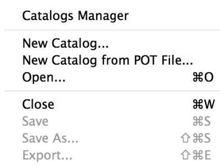


图27-1 新建POT文件

从这里，可以找到文件并选择它。要确保包含如下复数形式，如图27-2所示。

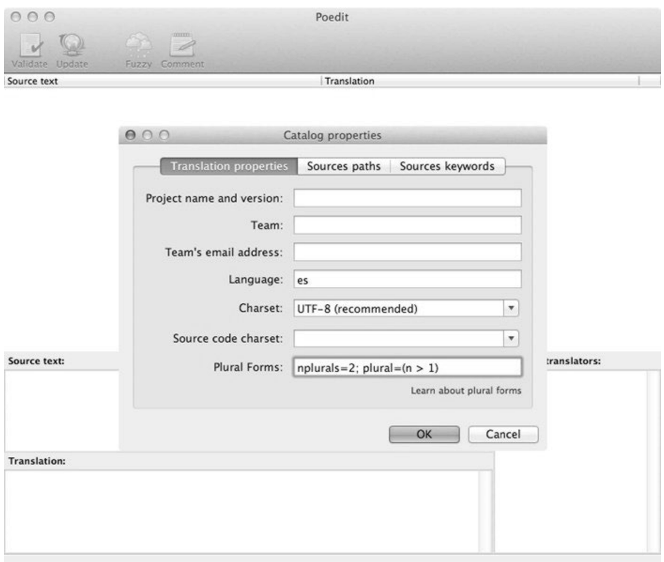


图27-2 设置复数形式

选择OK，然后你应该被带入一个显示要翻译的字符串的界面。然后应该针对特定的语言填

① http://www.poedit.net/

写这些字符串。

例如，如果想将应用翻译成西班牙语，就应该使用es语言。然后按照gettext的约定将它另存为es.po文件存储在templates.pot所在的目录，如图27-3所示。

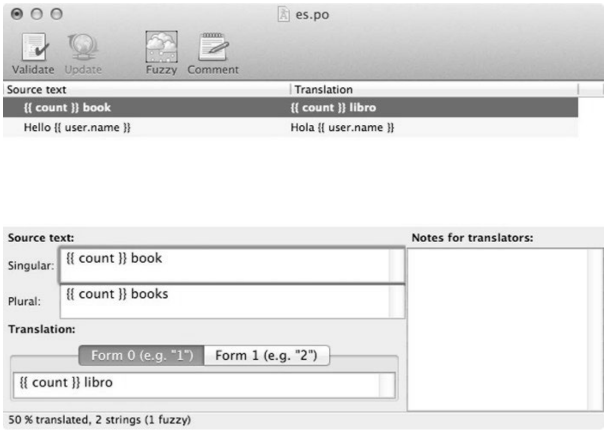


图27-3　翻译应用

完成编辑工作之后，可以保存这个文件，然后继续。

> 如果对应用做出了改变，可以简单地重新运行Grunt，然后在Poedit中选择“更新POT文件...”。这个步骤会更新新字符串，移除旧值，然后指出变化的部分，如图27-4所示。

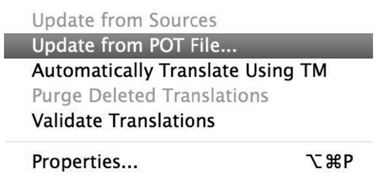


图27-4　翻译应用

27.12　编译新语言

最后，可以使用新的已编译好的语言格式来生成新的translation.js文件。

在此期间会多次使用Grunt将.pot文件编译为在运行时使用的`translation.js`文件。

这里还需要添加一个新的任务：`nggettext_compile`任务，这个任务会获取你的.pot文件，然后将它们包装为可以在应用中使用的语言。

基本的任务配置看起来像这样：

```
grunt.initConfig({
    nggettext_compile: {
        all: {
            files: {
```

```
                'app/scripts/translations.js': ['po/*.pot']
            },
        },
    }
})
```

这个配置会使用所有的po/*.pot文件生成app/scripts/translations.js文件。

也可以指定一个想要定义在translations内的特定模块，比如：

```
grunt.initConfig({
    all: {
            options: {
                module: 'myApp.translations'
            },
            files: {
                'app/scripts/translations.js': ['po/*.pot']
            },
        },
    }
})
```

建议设置一个Grunt任务来调用这两个nggettext_*函数，比如grunt.registerTask ('default', ['nggettext_extract', 'nggettext_compile']);。

现在，运行Grunt任务时，它会为我们生成translations.js文件。而我们只需要将这个文件包含在运行时的.html文件中，然后应用就已经做好翻译的准备了。

27.13 改变语言

至此我们已经设置好语言了，然后可以使用这些翻译来支持不同的语言。

gettext模块还包含一个叫做gettextCatalog的服务，这个服务可以注入到应用中设置当前语言。为了设置默认语言，可以简单地调用它：

```
angular.module('myApp')
.run(function(gettextCatalog) {
    gettextCatalog.currentLanguage = 'es';
});
```

上面这段代码会把应用程序的内容当作西班牙语加载。

注意，也可以在系统运行时做同样的事情，只需将gettextCatalog注入到Angular对象中，就像这样：

```
.controller('HomeController', function($scope, gettextCatalog) {
    $scope.user = { name: "Ari" };
    $scope.count = 1;
    $scope.changeLanguage = function() {
        gettextCatalog.currentLanguage = 'es';
    };
})
```

英文状态，如图27-5所示。



图27-5　英语

改变语言之后的西班牙语，如图27-6所示。



图27-6　西班牙语

第 28 章

缓　存

28

在大型的、互联网规模的Web应用中，限制从客户端调用API的能力使我们能够创建可伸缩的Web应用。

这不仅让前端呈现更快、更具响应性，还通过减少后端工作量的方式保护了后端。这样，后端就可以在前端服务于更多的用户。

28.1 什么是缓存

一个缓存就是一个组件，它可以透明地存储数据，以便未来可以更快地服务于请求。缓存不需要时常重新计算的数据是安全的，而重新获取数据会导致数据重复。

缓存能够服务的请求越多，整体系统性能就提升得越多。

传统的缓存服务器，比如Memcache，可以是为客户端内容提供服务的同一系统或者是远程系统。归根结底都是在服务器容量和流量这些选项之间作出选择。

根据内容的易变性，我们可以将精力集中于存储那些需要长期存储的缓存内容（比如，存储在磁盘中或者对于短期存储的内容就保存在内存中）。

缓存就像一个很大的键–值存储。一个键指向一块缓存的内容。当请求这部分内容时，如果在缓存中找到这个键并发现它有效（即命中缓存），那么就为这个请求提供相关内容。

如果没有这个键（即没找到缓存内容），那么缓存服务器就需要知道如何获取相关数据，存储它，然后为原始请求返回数据。

本章，我们将讨论Angular中的缓存策略，包括如何为服务器端内容设置memcache（轻量级的）以及使用Angular内置的缓存机制。它提供了一些很棒的库供我们使用。

28.2 Angular 中的缓存

Angular提供的内置缓存服务是一个很方便的特性，它让我们能够使用同一机制来缓存自定义内容。

28.2.1 `$cacheFactory` 简介

`$cacheFactory`是一个为所有Angular服务生成缓存对象的服务。在内部，`$cacheFactory`会创建一个默认的缓存对象，即使我们并没有显示地创建。

要创建一个缓存对象，可以使用$cacheFactory通过一个ID创建一个缓存：

```
var cache = $cacheFactory('myCache');
```

这里，定义了一个ID为myCache的缓存。这个$cacheFactory方法可以接受两个参数：

cacheId（字符串）：这个cacheId就是创建缓存时的ID名称。可以通过get()方法使用缓存名称来引用它。

options（对象）：这个选项用于指定缓存如何表现。一般情况下，这个选项对象是一个键：

- capacity（数字）

这个容量描述了在任何给定时间要使用缓存存储并保存的缓存键值对的最大数量。

$cacheFactory()方法返回一个缓存对象。

28.2.2　缓存对象

缓存对象自身有下列这些方法可以用来与缓存交互。

info()：info()方法返回缓存对象的ID、尺寸和选项。

put()：put()方法允许我们把任意JavaScript对象值形式的键（字符串）放进缓存中。

```
cache.put("hello","world");
```

put()方法会返回我们放入缓存中的值。

get()：get()方法让我们能够访问一个键对应的缓存值。如果找到了这个键，它会返回它的值；如果没有找到，它会返回undefined。

```
cache.get("hello");
```

remove()：remove()函数用于在找到一个键值对的情况下从缓存中移除它。如果没有找到，它就会返回undefined。

```
cache.remove("hello");
```

removeAll()：removeAll()函数用于重置缓存，同时移除所有已缓存的值。

destory()：destory()方法用于从$cacheFactory缓存注册表中移除指定缓存的所有引用。

28.3　$http 中的缓存

Angular的$http服务创建了一个带有ID为$http的缓存。（很意外，对吗？）要让$http请求使用默认的缓存对象很简单：$http()方法允许我们给它传递一个cache参数。

28.3.1　默认的$http缓存

当数据不会经常改变时，默认的$http缓存就特别有用了。可以像这样设置它：

```
$http({
    method: 'GET',
    url: '/api/users.json',
    cache: true
});
// 或者使用辅助方法.get()
$http.get('/api/user.json', {
    cache: true
});
```

现在，通过$http到URL/api/user.json的每个请求将会存储到默认的$http缓存中。这个$http缓存中的请求键就是完整的URL路径。

通过在$http选项中传入参数true，可以告诉$http服务使用默认的缓存。如果我们不想经常干扰那些缓存，使用默认缓存是很有用的。

然而，如果需要，也可以操作这个默认的$http缓存（比如，如果我们发起的另外一个没有缓存的请求提醒我们发生了增量变化，我们就可以在默认的$http请求中清除这个请求）。

为了引用$http的默认请求，只需通过$cacheFactory()使用ID来获取到该缓存：

```
var cache = $cacheFactory('$http');
```

对于所掌控的缓存，我们可以在需要时进行所有的正常操作，比如检索已缓存的响应，从缓存中清除条目，或者消除所有缓存的引用。

```
// 获取上一次请求的缓存
var usersCache = cache.get('http://example.com/api.users.json');
// 删除上一次请求的缓存入口
cache.remove('http://example.com/api.users.json');
// 重新开始并移除全部缓存
cache.removeAll();
```


尽管可以引用默认缓存，但是有时候能够对缓存有更多的控制以及针对缓存的表现创建规则可能更有用。这就需要创建一个新的缓存来使用$http请求。

28.3.2　自定义缓存

通过自定义的缓存来让$http请求发起请求很简单。可以采用传递缓存实例的方式，而不必传递一个布尔参数true给请求。

```
var myCache = $cacheFactory('myCache');
$http({
    method: 'GET',
    utl: '/api/users.json',
    cache: myCache
});
// 或者使用辅助方法.get()
$http.get('/api/users.json', {
    cache: myCahe
});
```

现在，$http将会使用自定义的缓存而非默认缓存。

28.4 为$http 设置默认缓存

尽管这很容易，但是每次我们想要发起一个$http请求时都要给它传递一个缓存实例并不方便，特别是对每个请求使用同一缓存的时候。

其实可以在模块的.config()方法中通过$httpProvider设置$http默认使用的缓存对象。

```
angular.module('myApp', []).config(function($httpProvider) {
    $httpProvider.defaults.cache = $cacheFactory('myCache', {capacity: 20});
    });
```

这个$http服务不再使用它为我们创建的默认缓存；它会使用我们自定义的缓存，实际上这就是一个近期缓存最久未使用算法[①]（Least Recently Used，LRU）。

LRU缓存根据缓存容量只保留最新的缓存数目。也就是说，我们的缓存容量为20，因此会缓存前20个请求，但是进入第21个请求时，最近最少使用的请求条目就会从缓存中被删除。这个缓存自身会负责具体哪些要维护，哪些要移除。

① LRU是一种高速缓存算法。基于这个算法丢弃最近最少使用的缓存。参考：维基百科：LRU（en.wikipedia.org/wiki/Least_Recently_Used#LRU）。——译者注

第 29 章

安全性

对于任何客户端应用而言，应该在构建时就考虑其安全性。然而在任何情况下都很难实现100%的保护，尤其是当客户端应用能看到全部代码的情况下，将更加困难。

在这一章，我们来看看一些保持应用程序安全的技术。我们将会看到如何掌握$sce服务，通过使用指令包装授权请求的方式，保障文本输入框的安全性（在讨论受保护的后端时）。

29.1 严格的上下文转义：$sce 服务

严格上下文转义模式（在Angular 1.2及更高版本中默认可用）用于告诉我们的应用，它需要绑定在某个上下文中，以便生成在该上下文中被标记为可信的值。

例如，当我们想要使用ng-bind-html绑定一个未加工的HTML给一个元素时，我们希望Angular使用这个HTML渲染该元素，而不是转义的文本。

```
<textarea ng-model="htmlBody"></textarea>
<div ng-bind-html="{{ htmlBody }}"></div>
```


$sce是一个非常出色的服务，它允许我们编写白名单，默认保护代码，并在很大程度上帮助我们防止XSS和其他漏洞。鉴于这种能力，理解它到底是什么很重要，这样我们才可以明智地使用它。

在上面的例子中，给<textarea>绑定了一个htmlBody模型。在这个textarea中，用户可以输入他们想在div中呈现的任意代码。例如，可能是编辑博客或者是评论时的实时预览，等等。

如果用户可以输入任意文本到这个文本字段中,这本质上相当于开启了一个巨大的安全漏洞。

默认情况下$sce服务会在所有插值表达式上为我们处理这一问题。不确定的字面量永远都是不可信的。

实质上，从Angular 1.2及更高版本的内置指令开始，$scope中的值就不再是绑定给它的值了，而是绑定$sce.getTrusted()方法的返回结果。

指令也使用新的$sce.parseAs()方法替代$parse服务监控属性绑定。$sce.parseAs()方法会在所有非恒定的字面量上调用$sce.$getTrusted()方法。

实际上，ng-bind-html指令会在幕后调用$sce.parseAsHtml()方法，然后将返回值绑定给DOM元素。ng-include指令以及任何定义在指令上的templateUrl都会执行这一行为。

启用这一特性时，所有内置指令都会自动调用$sce。当然，在自己的指令或者其他自定义组件中，也可以使用这一行为。

要设置$sce保护，需要注入$sce服务。

```
angular.module('myApp', [])
    .directive('myDirective', ['$sce', function($sce) {
        // 这里有权使用$sce服务
    }])
    .controller('MyController', ['$scope', '$sce', function($scope, $sce) {
        // 这里也有权使用$sce服务
    }]);
```

在上面的指令和控制器内，我们希望Angular能够允许受信任的内容回到视图内，同时接受可信任的插值输入。

$sce服务有一个简单的API，让我们能够设置和获取明确特定类型的可信任内容。

例如，要构建一个邮件预览程序。这个邮件客户端会允许用户在他们的邮件中编写HTML；而我们希望给他们的文本提供一个实时预览。

其中用到的HTML看起来可能像这样：

```
<div ng-app="myApp">
    <div ng-controller="MyController">
        <textarea ng-model="email.rawHtml"></textarea>
        <pre ng-bind-html="email.htmlBody"></pre>
```

现在，注意这里email上不同的属性：email.rawHtml和email.htmlBody会获取到<textarea></textarea>中的大段文本。在控制器内部，会将email.rawHtml解析为HTML，然后再输出到浏览器中。

在控制器内，可以设置一个$watch监控email.rawHtml的变化，然后无论什么时候发生变化都在HTML内容上运行一个受信任的解析器。

```
.controller('MyController', ['$scope', '$sce', function($scope, $sce) {
    // 在email.rawHtml上设置监控
    $scope.$watch('email.rawHtml', function(v) {
        // 假设在非编译($compile)阶段
        if(v) {
            // 将htmlBody渲染为受信任的HTML
            $scope.email.html Body = $sce.trustAsHtml($scope.email.rawHtml);
        }
    });
}]);
```

现在，每当email.rawHtml的内容发生变化时，都会在这个内容上执行一个解析器，然后取回适当的HTML内容。而这个内容会被渲染为干净的HTML，也就是应用程序的安全来源。

假如还希望页面上能执行用户编写的自定义JavaScript会怎样呢？例如,希望允许用户编写含自定义JavaScript的电子贺卡，那么就要允许他们在页面让运行自定义的JavaScript代码。

要用到的HTML看起来可能像这样：

```
<textarea ng-model="email.rawJs"></textarea>
<pre ng-bind="email.jsBody"></pre>
<button ng-click="runJs()">Run</button>
```

在这段代码中，其运行原理与解析原始文本为安全文本一致。这次，还要添加第三个元素，也就是在作用域中调用runJs()的按钮。

正如在HTML绑定中看到的，我们会监控这个JavaScript代码片段：

```
.controller('MyController', ['$scope', '$sce', function($scope, $sce) {
    // 在email.rawJs上设置监控
    $scope.$watch('email.rawJs', function(v) {
        if(v) {
            $scope.email.jsBody = $sce.trustAsJs($scope.email.rawJs);
        }
    });
}]);
```

注意，这次我们没有使用trustAsHtml，而是使用trustAsJs()方法。这个方法会告诉Angular将指定的文本解析为可执行的JavaScript代码。调用结束后，这会得到一个可以在应用程序上下文中使用eval()执行的、安全的、已解析的JavaScript代码片段。

现在可以将runJs()方法设置给用户，然后可以应用email.rawJS来运行这个代码片段。

```
// ...
$scope.runJs = function() {
    eval($scope.email.jsBody.toString());
};
```

对于在JavaScript代码片段上执行eval，还有更智能的方法。但是不推荐在产品中使用eval。

Angular中还提供了内置的保护机制：只能从加载应用的同一域中以及应用所在的协议内加载模板。Angular通过在templateUrl上强制调用$sce.getTrustedResourceUrl来对它进行保护。

这个协议不能替代浏览器的同源策略以及跨域资源共享（CORS）。这些策略仍然能够有效保护浏览器。

29

但是可以通过使用$sceDelegateProvider设置域白名单或者黑名单的方式来改写这个值。

29.2 URL 白名单

在模块的config()函数内可以设置新的白名单和黑名单。

```
angular.module('myApp', [])
    .config(['$sceDelegateProvider', function($sceDelegateProvider) {
        // 设置一个新的白名单
        $sceDelegateProvider.resourceUrlWhitelist(['self']);
}]);
```

可以使用resourceUrlWhitelist()方法设置新的白名单。这个函数接受一个可选的参数。

- 白名单列表（数组）。

如果没有传入参数，那么这个函数就作为一个getter方法，同时返回当前设置的白名单数组。

如果传入了白名单参数，resourceUrlWhitelist就会使用新的数组替换原来的数组。

每个数组元素必须是一个正则表达式或者是字符串'self'。当设置为'self'时，Angular会

确保所有的URL都只匹配与应用所在域一致的URL。使用一个正则表达式时，Angular会匹配与测试资源对应的绝对URL。

如果这个数组为空，$sce会阻塞所有的URL。

使用'self'时可以在HTML文档中使用https协议的资源。

为了启动每个独立的URL，每个域都要加入白名单：

```
angular.module('myApp', [])
    .config(['$sceDelegateProvider',
    function($sceDelegateProvider) {
        // 设置新的白名单
        $sceDelegateProvider.resourceUrlWhitelist(['.*']);
    }]);
```

默认情况下，白名单被设置为['self']。

29.3　URL 黑名单

也可以使用黑名单URL替换白名单。它通常比依靠白名单更安全,但是你可以结合使用它们。对于可信任的域而言，白名单很有用；黑名单通常服务于域名重定向操作。

可以使用resourceUrlBlacklist()方法设置新的黑名单。这个方法也接受一个可选的参数。

- 黑名单列表（数组）。

如果没有传入参数，这个函数就会返回当前设置的黑名单数组。

如果传入了黑名单参数，新的数组就会替换原来的黑名单。

黑名单数组中的每个元素必须是一个正则表达式或者是字符串'self',尽管在使用黑名单的情况下，它没什么用。但是使用正则表达式时，它匹配与测试资源相对的绝对URL。

对于可信任的内容，黑名单总是最后才决定什么是可接受的，什么是不可接受的。

默认情况下，黑名单被设置为一个空数组[ ]。

29.4　$sce API

$sce库中有两个我们会用到的主要函数，以及一系列辅助函数。

29.4.1　getTrusted

要获取一个特定类型的可信任版本的值，可以调用getTrusted()方法。

这个getTrusted()方法接受两个参数。

- 类型（字符串）。

这个字符串代表使用该值的上下文类型。对于可用类型可以查看sce类型列表。

- maybeTrusted

这个值是从$sce.trustAs返回的值。如果无效，它抛出一个异常。

$sce库还有一些适用于getTrusted()方法的辅助方法。

下列方法的调用从功能上讲是等价的：

getTrustedCss(value)	getTrusted($sce.CSS, value)
getTrustedHtml(value)	getTrusted($sce.HTML, value)
getTrustedJs(value)	getTrusted($sce.JS, value)
getTrustedResourceUrl(value)	getTrusted($sce.RESOURCE_URL, value)
getTrustedUrl(value)	getTrusted($sce.URL, value)

29.4.2 parse

类似于$parse服务，parse方法用于将Angular表达式转换为函数。如果表达式是一个恒定的字面量，它就调用$parse服务；否则，调用$sce.getTrusted()服务。

parse()方法接受两个参数。

- 类型（字符串）。

这个类型代表使用该值的$sce上下文类型。对于可用类型可以查看sce类型列表。

- 表达式（字符串）。

Angular要编译的表达式。

parse()方法返回一个：function(context, locals)形式的函数：

- context（对象）。

这个对象表示表达式应该在哪里被求值。通常是一个$scope对象。

- locals（对象）。

局部变量，在context中重写值时非常有用。

$sce库有一些适用于parse()方法的辅助方法。

下列方法的调用从功能上讲是等价的：

parseAsCss(expr)	parseAs($sce.CSS, expr)
parseAsHtml(expr)	parseAs($sce.HTML, expr)
parseAsJs(expr)	parseAs($sce.JS, expr)
parseAsResourceUrl(expr)	parseAs($sce.RESOURCE_URL, expr)
parseAsUrl(expr)	parseAs($sce.URL, expr)

29.4.3 trustAs

trustAs()方法返回一个对象，Angular信任该对象，可以在特定的严格上下文转义环境中使用它。比如ng-bind-html和ng-include绑定，使用给它们提供的值。

这个trustAs()方法接受两个参数。

- 类型（字符串）。

这个$sce上下文类型表示哪里的值是安全的。对于可用类型可以查看sce类型列表。

- 值。

这个值表示我们可以用它来替代提供的值。

trustAs()方法返回一个值，可以用于Angular期望$sce.trustAs()返回值的地方。

$sce库还有一些适用于trustAs()方法的辅助方法。

下列方法的调用从功能上讲是等价的：

trustAsHtml(value)	trustAs($sce.HTML, value)
trustAsJs(value)	trustAs($sce.JS, value)
parseAsResourceUrl(value)	trustAs($sce.RESOURCE_URL, value)
trustAsUrl(value)	trustAs($sce.URL, value)

29.4.4　isEnabled

isEnabled()方法没有参数，同时它返回一个布尔值，该值告诉我们是否启用了sce环境。如果启用了，它返回true；否则返回false。

29.5　配置$sce

如果想在运行应用时完全禁用sce子系统（虽然不鼓励这么做，默认情况下它还是提供了安全性保护），可以在应用的config()函数中像这样禁用它：

```
angular.module('myApp', [])
    config(['$sceProvider', function($sceProvider) {
        // 关闭SCE
        $sceProvider.enable(false);
    }]);
```

29.6　可信赖的上下文类型

$sce库默认情况下支持五个内置的上下文类型。Angular使用这些上下文类型解析和确定一个上下文与另一个上下文中什么是安全的。

上　下　文	描　　述
$sce.HTML	告诉Angular，在应用中这是安全HTML来源
$sce.CSS	告诉Angular，在应用中这是安全的CSS来源
$sce.URL	告诉Angular，这个带URL的链接是安全的
$sce.RESOURCE_URL	告诉Angular，这个带URL的链接是安全的，同时引入应用的内容也是安全的
$sce.JS	告诉Angular，在应用程序中这是可以安全执行的内容

第 30 章 AngularJS和IE浏览器

最新版本的Angular（1.3.0）[①]减少了对IE8的支持。本章是本书特意为1.2.x版留下的。

AngularJS可以无缝地运行在大多数现代浏览器中，在Safari、Google Chrome、Google Chrome Canary以及FireFox中都可以很好地工作，但在臭名昭著的IE8以及更低版本中则可能会有些问题。

> 欲了解更多信息，请阅读AngularJS文档中的IE使用指南[②]。

如果你计划为IE8及更低版本的IE浏览器发布应用程序，则需要额外注意一下，以便支持它们。

IE浏览器不希望元素名以ng开头：因为它会认为这个前缀是一个XML命名空间。IE浏览器会忽略这些元素，除非这些元素有一个正确的命名空间声明：

```
<html xmlns:my="ignored">
```

> 这个xmlns:ng="http://angularjs.org"会让IE更好地工作。

如果希望IE能够识别非标准的HTML标签，需要在文档头部创建好这些标签。在文档head中可以这样做。

```
<!doctype html>
<html xmlns:ng="http://angularjs.org">
<head>
<!--[if lte IE 8]
<script>
    document.createElement('ng-view');
    // 其他自定义元素
</script>
<![endif]-->
</head>
<body>
<!-- ... -->
```

推荐使用属性（attribute）形式的指令，这样就无需创建自定义元素来支持IE：

```
<div data-ng-view></div>
```

① 本书出版时，Angular 1.3还处于beta版阶段，最新版仍为1.2.x。——译者注

② http://docs.angularjs.org/guide/ie

为了让AngularJS能在IE7及更早版本中工作，还需要一个JSON.stringify polyfill①。可以使用JSON3②或者JSON2③实现。

我们需要根据浏览器的类型来引入这个文件。首先需要下载这个文件，将它存储在应用程序的某个地方，最后在头部引用它，就像这样：

```
<!doctype html>
<html xmlns:ng="http://angularjs.org">
<head>
<!--[if lte IE 8]
<script src="lib/json2.js"></script>
<![endif]-->
</head>
<body>
<!-- ... -->
```

此外，为了在IE中使用ng-app指令，还要设置元素的id为ng-app。

```
<body id="ng-app" ng-app="myApp">
<!-- ... -->
```

还可以利用angular-ui-utils库的ie-shiv模块帮助我们在DOM中提供自定义元素。

为了使用ui-utils的ie-shiv库，需要确保安装了angular-ui库。如果下载了ui-utils库，同时引入了这个模块，那么安装起来就很容易了。可以在Github上的https://github.com/angular-ui/ui-utils④中找到这个ui-utils。

先来确保在应用可访问的位置包含了ui-utils模块，然后像这样引入这个文件：

```
<!--[if lte IE 8]>
<script type="text/javascript">
// 在这里定义自定义指令
</script>
<script src="lib/angular-ui-ieshiv.js"></script>
<![endif]-->
```

在这里，我们仅仅在IE8及更早版本的IE中激活了ie-shiv。这个shiv允许我们在全局作用域上添加自定义指令，它会为IE创建适当的声明。

这个shiv库会查找window.myCustomTags对象。如果定义了window.myCustonTags，这个库会在加载时引入这些标签，同时一同引入其余的Angular库指令：

```
<!--[if lte IE 8]>
<script type="text/javascript">
// 在这里定义自定义指令
window.myCustomTags = ['myDirective'];
</script>
<script src="lib/angular-ui-ieshiv.js"></script>
<![endif]-->
```

① polyfill用来添加一些原生功能支持的功能。更多信息可以参考“[译]shim和polyfill有什么区别”（www.cnblogs.com/ziyunfei/archive/2012/09/17/268829.html）和“HTML5逸事：一袋‘腻子粉’的故事（待续）”（www.ituring.com.cn/article/766）。——译者注

② http://bestiejs.github.io/json3/

③ https://github.com/douglascrockford/JSON-js

④ https://github.com/angular-ui/ui-utils

30.1　Ajax 缓存

常见的浏览器里，IE是唯一缓存XHR请求的。为了避免这一缺陷，可以在HTTP响应头中设置Cache-Control值为no-cache。

这是现代浏览器的默认行为，使用它可以为IE用户提供更好的体验。

可以像这样修改每个请求的默认HTTP头：

```
.config(function($httpProvider) {
    $httpProvider.defaults.headers.common['Cache-Control'] = 'no-cache';
});
```

30.2　AngularJS 中的 SEO

诸如Google和Bing这样的搜索引擎，都抓取静态Web页面，而非含有大量JavaScript的客户端应用。搜索引擎机器人通常都快速、高效地抓取数据，因此大多数情况下在抓取页面时不会渲染JavaScript。

这是因为这些JavaScript应用都需要一个JavaScript引擎来解释它，例如PhantomJS或者v8。Web爬虫程序通常在加载Web页面时并不会使用JavaScript解释器。

> 在搜索引擎中不包含JS解释器有充分的理由：因为它们不需要，这样做只会在抓取Web页面时更慢、更低效。

30.3　使 Angular 应用可被索引

有几种不同的方式可以让Google处理应用索引。一个常见的做法是使用后端程序为你的Angular应用提供服务。这种方法的优点是实现简单，不会有太多重复的代码。

第二种做法是在JavaScript中的<noscript>标签内渲染所有由Angular应用交付的内容。稍后会简单讨论这一主题，因为实现<noscript>方法时很大程度上取决于如何交付应用。例如，可以在服务端渲染页面时使用<noscript>。

30

30.4　服务端

Google和其他高级搜索引擎都支持hashbang格式的URL，我们可以用它来识别当前要访问的页面。搜索引擎会将这个URL转换为一个自定义的URL格式，以便服务器可以访问它们。

搜索引擎访问这个URL，并期待获取到浏览器将要接收的HTML（完全渲染过的HTML内容）。例如，Google会把hashbang格式的URL：

```
http://www.ng-newsletter.com/#!/signup/page
```

转变为URL：

```
http://www.ng-newsletter.com/?+escaped_fragment_=/signup/page
```

在我们的Angular应用中，需要根据我们所使用的URL风格让Google以不同的方式处理应用站点。

30.4.1　hashbang语法

Google最初编写Ajax采集规范就是为了使用hashbang语法传送URL，这是一个为JS应用程序创建永久链接的原始方法。

这需要在应用路由中使用`hashPrefix`（默认的）配置我们的应用：

```
angular.module('myApp', [])
.configure(['$location', function($location) {
    $location.hashPrefix('!');
}]);
```

30.4.2　HTML5 路由模式

新的HTML5 pushState并不以相同的方式工作：它会修改浏览器的URL和历史记录。为了让Angular应用“欺骗”搜索机器人，可以在header中添加一个简单的元素：

```
<meta name="fragment" content="!">
```

这个元素会让Google蜘蛛使用新的爬行规范来抓取你的站点。当它遇到这个标签时，它会使用`?_escaped_fragment=`标签重新访问站点，而不是采用标准的抓取站点的方式。

假设要在`$location`服务中使用HTML5模式，可以像这样设置页面以使用`httml5Mode`：

```
angular.module('myApp', [])
.configure(['$routeProvider', function($routeProvider) {
    $routeProvider.html5Mode(true);
}]);
```

查询字符串中有了`_escaped_fragment_`后，我们可以让后端服务器提供静态的HTML而不是客户端应用。

现在，后端服务器可以检测请求中是否包含`_escaped_fragment_`字段，如果包含，我们将提供静态HTML而不是纯Angular应用。

> 还可以使用代理实现这个功能，比如Apache或者Nginx，或者是一个后端服务。如何设置超出了本书的范畴，然而，我们会使用一个NodeJS应用来讨论如何设置它们。

30.5　服务端处理 SEO 的选项

有许多选项可以使我们的站点更好地支持搜索引擎优化（Search Engine Optimization，SEO）。我们将会使用三种不同的方式演示如何从服务器端交付应用：

- 使用Node/Express中间件；
- 使用Apache重写URL；
- 使用Ngnix代理URL。

30.5.1 使用Node/Express中间件

尽管在这个例子中我们使用的是NodeJS，但这一实现只是使用后端提供静态HTML的一种方式。无论你使用什么样的后端，这一方案都是可行的。

为了使用NodeJS和Express（基于NodeJS的Web应用程序框架）交付静态HTML，我们必须添加一些用来在查询参数中查找_escaped_fragment_的中间件：

```
// 在你的app.js配置中共享创意
app.use(function(req, res, next) {
    var fragment = req.query._escaped_fragment_;

    if(!fragment) return next();

    // 如果fragment为空，则服务于首页
    if(fragment === "" || fragment === "/")
        fragment = "/index.html";

    // 如果fragment不是以'/'开始的，则将'/'前置插入fragment
    if(fragment.charAt(0) !== "/")
        fragment = '/' + fragment;

    // 如果fragment不是以'.html'结尾的，则将它插入fragment中
    if(fragment.indexOf('.html') == -1)
        fragment += ".html";

    // 服务于静态html快照
    try {
        var file = __dirname + "/snapshots" + fragment;
        res.sendfile(file);
    } catch (err) {
        res.send(404);
    }
});
```

这个中间件认为我们的快照存放在叫做“/snapshots”的顶级目录中，然后会基于请求路径为文件提供服务。

例如，当请求/时，它会提供index.html；当请求为/about时，它会提供snapshots目录中的about.html。

30.5.2 使用Apache重写URL

如果使用Apache服务器[①]交付Angular应用，那么可以添加几行代码到配置中，用来提供快照服务而不是JavaScript应用。

可以使用mod_rewrite模块来检测路由请求中是否包含_escaped_fragment_查询参数。如果确实包含，它会重写请求，以指向/snapshots目录中的静态版本的文件。

要想使用重写机制，需要启用适当的模块：

```
$ a2enmod proxy
$ a2enmod proxy_http
```

① http://httpd.apache.org/

然后需要重新载入Apache配置：

```
$ sudo /etc/init.d/apache2 reload
```

可以在站点的虚拟主机配置中，或者位于服务器根目录的.htaccess文件中设置重写规则。

```
RewriteEngine On
Options +FollowSymLinks
RewriteCond %{REQUEST_URI} ^/$
RewriteCond %{QUERY_STRING} ^_escaped_fragment_/?(.*)$
RewriteRule ^(.*)$ /snapshots/%1? [NC,L]
```

30.5.3　使用Ngnix代理URL

如果使用Nginx[1]为Angular应用提供服务，并且在查询字符串中有一个_escaped_fragment_参数，那么也可以添加一些配置以便提供应用的快照。

和Apache不同，Nginx无需启用什么模块，因此可以简单地更新配置来处理替换文件路径的问题。

在Nginx配置文件中（比如/etc/nginx/nginx.conf），需要确保配置信息看起来像这样：

```
server {
    listen: 80;
    server_name example;

    if($args ~ "_escaped_fragment_=/?(.+)") {
        set $path $1;
        rewrite ^ /snapshots/$path last;
    }

    location / {
        root /web/example/current/;
        # Comment out if using hash urls
        if(!-e $request_filename) {
            rewrite ^(.*)$ /index.html break;
        }
        index index.html;
    }
}
```

这一步完成之后就可以重新加载配置信息了：

```
sudo /etc/init.d/nginx reload
```

30.6　获取快照

我们可以使用PhantomJS[2]或者zombie.js[3]等工具来渲染页面，以便提供后台应用的HTML快照。当Google使用_escaped_fragment_查询参数请求一个页面时，我们就可以简单地返回并渲染这个页面。

① http://wiki.nginx.org/Main
② http://phantomjs.org/
③ http://zombie.labnotes.org/

下面我们会讨论两个获取快照的方法：使用Zombie.js和一个Grunt工具。这里不会涉及使用出色的PhantomJS①，因为已经有很多介绍它的优秀资源了。

30.7 使用 Zombie.js 获取 HTML 快照

要使用Zombie.js②，需要安装zombie这个npm包：

```
$ npm install zombie
```

现在，我们可以通过使用zombie来使用NodeJS保存文件了。首先，在这个过程中会用到一些辅助方法：

```
var Brower = require('zombie'),
    url = require('url'),
    fs = require('fs'),
    saveDir = __dirname + '/snapshots';

var scriptTagRegExp = /<script\b[^<]*(?:(?!<\/script>)<[^<]*)*<\/script>/gi;

var stripScriptTags = function(html) {
    return html.replace(scriptTagRegexp, '');
}

var browserOpts = {
    waitFor: 2000,
    loadCSS: fase,
    runScripts: true
}

var saveSnapshot = function(uri, body) {
    var lastIdx - uri.lastIndexOf('#/');
    if(lastIdx < 0) {
        // 如果使用html5mode
        path =
        url.parse(uri).pathname;
    } else {
        // 如果使用hashbang模式
        path = uri.substring(lastIdx + 1, uri.length);
    }

    if(path === '/') path = "/index.html";

    if(path.indexOf('.html') == -1)
        path += ".html";

    var filename = saveDir + path;
    fs.open(filename, 'w', function(e, fd) {
        if(e) return;
        fs.write(fd, body);
    });
};
```

现在，我们需要做的就是遍历所有页面，将每个链接从相对链接转换为绝对链接（这样爬虫才能追踪它们），然后保存生成的HTML。

① http://phantomjs.org/

② http://zombie.labnotes.org/

上面的浏览器配置中设置了一个相对较大的waitFor。这个选项会覆盖我们所关心的90%的情况。如果想在获取一个快照时使用更精确的方式，而不是等待2秒，则需要修改Angular应用来触发一个事件，然后在Zombie浏览器中监听这个事件。

由于我们希望尽可能自动化，而不想污染Angular代码，因此宁愿设置一个相对较高的超时来尝试让代码保持稳定。

crawlPage()函数：

```
var crawlPage = function(idx, arr) {
    // location = window.location
    if(idx < arr.length) {
        var uri = arr[idx];
        var brower = new Brower(browerOpts);
        var promise = brower.visit(uri).then(function() {
            // 将链接转换为绝对链接，然后保存它们
            // 如果需要并且还没准备好就抓取它们
            var links = brower.queryAll('a');
            links.forEach(function(link) {
                var href = link.getAttribute('href');
                var absUrl = url.resolve(uri, href);
                link.setAttribute('href', absUrl);
                if(arr.indexOf(absUrl) < 0) {
                    arr.push(absUrl);
                }
            });

            // 保存
            saveSnapshot(uri, brower.html());
            // 在下次迭代中再次调用
            crawlPage(idx+1, arr);
        });
    }
}
```

现在，可以在我们的页面中调用这个方法了：

```
crawlPage(0, ["http://localhost:9000"]);
```

30.8　使用 grunt-html-snapshot

我们推荐使用Grunt工具grunt-html-snapshot来生成快照。因为我们使用Yeoman[①]，所以Grunt已经存在于我们的构建过程之中，我们只需设置这个任务在发布应用之后运行即可。

要安装grunt-html-snapshot，可以像这样使用npm安装它：

```
npm install grunt-html-snapshot --save-dev
```

如果不使用Yeoman[②]，就需要将这个任务作为一个Grunt任务包含到Gruntfile.js中：

```
grunt.loadNpmTasks('grunt-html-snapshot');
```

① http://yeoman.io/

② http://yeoman.io/

设置好这个任务之后，可以对我们的站点做一些配置。为了设置这些配置，需要在Gruntfile.js中创建如下新配置块：

```
htmlSnapshot: {
    debug: {
        options: {}
    },
    prod: {
        options: {}
    }
}
```

现在就可以为不同的阶段添加首选项了：

```
htmlSnapshot: {
    debug: {
        options: {
            snapshotPath: 'snapshot/',
            sitePath: 'http://127.0.0.1:9000/',
            msWaitForPages: 1000,
            urls: [
                '/',
                '/about'
            ]
        }
    },
    prod: {
        options: {}
    }
}
```

要查看所有可用的配置选项列表，可参考如下文档：https://github.com/cburgdorf/grunt-html-snapshot。

30.9 Prerender.io

此外，我们还可以使用开源的工具，比如Prerender.io[①]，它包含一个在运行时渲染站点的Node服务器，和一个与后端通讯在运行时预渲染HTML的Express中间件。

本质上，prerender.io接受一个URL，然后返回渲染后的HTML（不带script标签）。实质上，它会从应用中像这样调用我们部署的预渲染服务器：

```
GET http://our-prerenderserver.com/ http://localhost:9000/#/about
```

这个GET会返回`#/about`页面渲染后的内容。

设置一个prerender集群实际上很容易做到。此外，我们还会展示如何将自己的预渲染服务器集成到Node应用中。Prerender.io还可以通过一个叫做`prerender_rails`的gem支持Ruby on Rails，但是在这里我们不会介绍如何设置它。

设置自己的服务器来运行它相当容易。只需运行`npm install`来安装依赖，然后通过Foreman或者Node运行以下命令就行了：

① http://prerender.io/

```
$ npm install
$ node index.js
# 或者使用foreman
$ foreman start
```

prerender库还可以很方便地运行在heroku上：

```
$ git clone http://github.com/collectiveip/prerender.git
$ heroku create
$ git push heroku master
```

这里我们将渲染后的HTML存储在S3[①]中，因此推荐你使用内置的S3缓存。可以阅读文档[②]来了解如何设置这个缓存。

在服务器运行之后，只需要将这个抓取程序集成到应用中就行了。在Express中，使用Node库prerender-node集成它非常容易。

为了安装prerender-node，我们再次使用npm：

```
$ npm install --save prerender-node
```

安装好这个库之后，让Express应用使用这个中间件即可：

```
var prerender =
require('prerender-node')
.set('prerenderServiceUrl',
     'http://our-prerenderserver.com/');
app.use(prerender);
```

就是这样！最后这个片段用于告诉Express应用，如果它察觉到爬虫请求（通过定义_escaped_fragment_或者用户代理字符串），就应该在合适的URL上为prerender服务创建一个GET请求，然后获取这个页面预渲染好的HTML。

30.10 <noscript>方法

还可以使用<noscript>标签来渲染页面，而无需使用后端服务器。不幸的是，对于所有页面都使用这个方法是非常复杂的，因为需要从<noscript>外部将页面的所有元素复制到这个标签中。

这样做会很麻烦，为了保持两部分同步要做大量的工作，因此我们不推荐使用这个没有构建工具协助的方法。

① S3就是Amazon S3，全程亚马逊简易储存服务，这是亚马逊公司提供的一个网络存储服务。更多信息参考维基百科：Amazon S3（zh.wikipedia.org/wiki/Amazon_S3）。——译者注

② https://github.com/collectiveip/prerender#s3-html-cache

第 31 章

构建Angular Chrome应用

Chrome[①]浏览器是Google定制的浏览器。令人难以置信的不仅是它的速度，它更领跑在Web开发的最前沿，它为前端提供了在线和离线Web体验。

Chrome应用就是嵌入在Web浏览器中运行的应用程序，但是它旨在提供原生应用的感觉。由于它们本身是运行在Chrome中的，因此可以使用HTML5、JavaScript和CSS3来编写它们，此外它们还能够访问类原生的功能，但它们并不是真正的Web应用程序。

Chrome应用可以使用Chrome相关API和服务，而且还能为用户提供集成桌面应用的体验。

Chrome应用和Web应用之间一个更有趣的区别是，Chrome应用是在本地加载的，因此能够立即呈现，而无需等待网络将组件完全下载下来。这一特性大大提升了运行应用时的性能和用户体验。

31.1 了解 Chrome 应用

让我们先来深入观察一下Chrome应用是如何工作的，以及如何开始构建自己的Chrome应用。

每个Chrome应用程序都有三个核心文件。

31.1.1 manifest.json

这个manifest.json文件用于描述应用程序的元数据，比如名称 、描述、版本以及如何启动这个应用程序。

31.1.2 背景脚本

这个背景脚本用于设置应用程序如何响应系统级的事件，比如用户安装你的应用或者启动它，等等。

31.1.3 视图

大多数的Chrome应用程序都有一个视图。这个组成部分是可选的，但是它通常对我们应用程序而言非常有用。

① https://www.google.com/intl/en/chrome/browser/

31.2　构建你的Chrome应用

这一节将使用Angular创建一个高级Chrome应用程序。创建一个由Rainfall[①]团队创建的非常出色的Chrome Web应用Currently的克隆版，如图31-1所示。



图31-1　Currently

接下来构建一个叫做Presently的克隆版。

Presently架构

在构建Presently时，需要考虑应用程序的架构。这样做可以在开始编写代码时就深刻理解该如何构建这个应用。

和Currently一样，Presently也是一个“newtab”应用，这意味着它会在每次打开一个新的标签页时启动。

Presently有两个主要界面：

- 主界面

这个界面显示当前时间和天气。它还会在天气旁边显示一些天气图标。

- 设置界面

这个界面允许用户改变应用的地理位置。

为了支持主界面，需要能够显示正确的日期和时间，以及从远程API服务中获取天气数据。

为了支持设置界面，我们需要整合远程API服务，以便使输入框可以自动提示可能的城市。

最后，我们将会使用基本的本地存储功能（session存储）保存整个应用的设置。

31.3　搭建框架

我们将会设置一个像这样的文件结构来构建这个应用，如图31-2所示。

① http://blog.rainfalldesign.com/

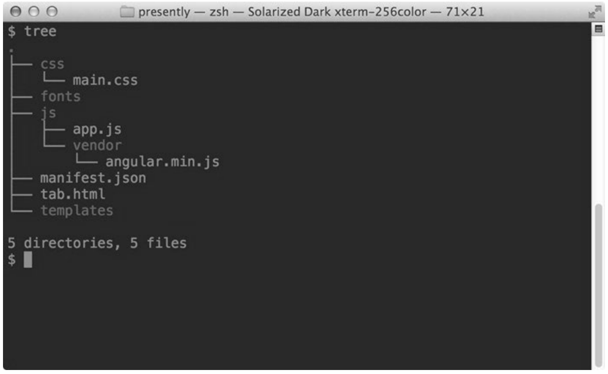


图31-2 文件结构

这里把CSS文件存放到css/目录中，自定义字体存放在fonts/中，而JavaScript文件存放在js/中。还会设置js/app.js文件为主JavaScript文件，以及根目录中的tab.html为应用的HTML模板。

> 有一些非常好的工具可以帮助我们创建Chrome应用扩展的引导程序，比如Yeoman[1]。

在启动Chrome扩展之前，还需要先获取一些依赖。

这里将会从angularjs.org[2]下载最新版的angular.min.js[3]，以及angular-route.min.js[4]，然后将它们保存到js/vendor/目录中。

最后，这里还会使用Twitter的Bootstrap 3框架为我们应用提供样式，因此还需要从getbootstrap.com[5]下载bootstrap.min.css文件，然后将它保存到css/中。

> 在产品中，有多个开发人员一起工作时，使用Bower[6]这类工具通常能够更有效地管理依赖。由于这里要构建一个newtab应用，为了快速启动而使应用程序保持轻量级是很重要的。

31.4 manifest.json

在我们编写的每个Chrome应用中，都需要设置一个`manifest.json`文件。这个manifest文件用于告诉Chrome应用程序该应该如何运行，应该使用什么文件以及有什么权限，等等。

在这里，这个manifest.json文件需要将我们的应用描述为一个newtab应用，还要描述Chrome应用需要的`content_security_policy`（这是一个描述应用程序能做什么不能做什么的策略选项）和背景脚本。

① http://yeoman.io/
② http://angularjs.org/
③ http://code.angularjs.org/1.2.16/angular.min.js
④ http://code.angularjs.org/1.2.16/angular-route.min.js
⑤ http://getbootstrap.com/
⑥ http://bower.io/

```
{
    "manifest_version": 2,
    "name": "Presently",
    "description": "A currently clone",
    "version": "0.1",
    "permissions": ["http://api.wunderground.com/api/"],
    "background": {
        "scripts": ["js/vendor/angular.min.js"]
    },
    "content_security_policy": "script-src 'self'; object-src 'self'",
    "chrome_url_overrides": {
        "newtab": "tab.html"
    }
}
```

manifest.json中的键相对而言比较简单，并且它允许设置名称、manifest版本和应用版本，等等。为了让Chrome将应用作为newtab应用启动，在这里设置这个应用覆盖所有新标签页。

31.5　tab.html

这个应用程序的主HTML文件就是tab.html文件。当我们在Chrome打开一个新标签页时就会加载这个文件。

我们将在这个tab.html文件内建立这个基本的Angular应用：

```
<!doctype html>
<html data-ng-app="myApp" data-ng-csp="">
    <head>
        <meta charset="UTF-8">
        <title>Presently</title>
        <link rel="stylesheet" href="css/bootstrap.min.css">
        <link rel="stylesheet" href="css/main.css">
    </head>
    <body>
        <div class="container">
        </div>
        <script src="./js/vendor/angular.min.js"></script>
        <script src="./js/vendor/angular-route.min.js"></script>
        <script src="./js/app.js"></script>
    </body>
</html>
```

这是Angular应用程序非常基本的结构，看起来几乎与任何Angular应用相同，除了`data-ng-csp=""`之外。

这个`ngCsp`会启用内容安全策略（Content Security Policy，CSP）来支持我们的应用。由于Chrome应用会阻止浏览器使用`eval`或者`function(string)`生成的函数，而Angular要使用`function(string)`生成的函数来加速，`ngCsp`可以让Angular能够计算所有表达式的值。

然而，这个兼容模式出于考虑性能的成本，在执行操作时比较慢，但是在这个过程中它不会抛出任何安全隐患。

CSP还禁止使用JavaScript引入内联样式表规则，因此你需要手动引入angular-csp.css。

可以在http://code.angularjs.org/snapshot/angular-csp.css[1]找到angular-csp.css文件。

最后，还必须把ngCsp放在Angular应用的根元素旁边：

```
<htmlng-appng-csp>
```

> 在没有ngs-csp指令的情况下，Chrome应用将不会运行；它会抛出一个安全隐患。如果看到安全隐患被抛出，请检查应用的根元素以确保有这个指令。

31.6　在 Chrome 中加载应用

对于正在开发中的应用也可以将它加载到Chrome以便在浏览器中了解进展情况。要在Chrome中加载应用，首先应该导航到URL：chrome://extensions/。

导航到这里之后，可以点击“加载正在开发的扩展程序...”按钮，然后找到项目的根目录（这个目录中包含在上面定义的manifest.json文件），如图31-3所示。

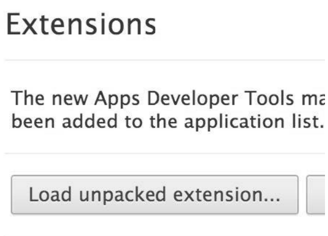


图31-3　加载未打包的扩展

这个应用程序加载到Chrome浏览器之后，我们可以打开一个新标签页，并且应该看到带有一个错误信息的空应用（不要担心，很快就会修复它），如图31-4所示

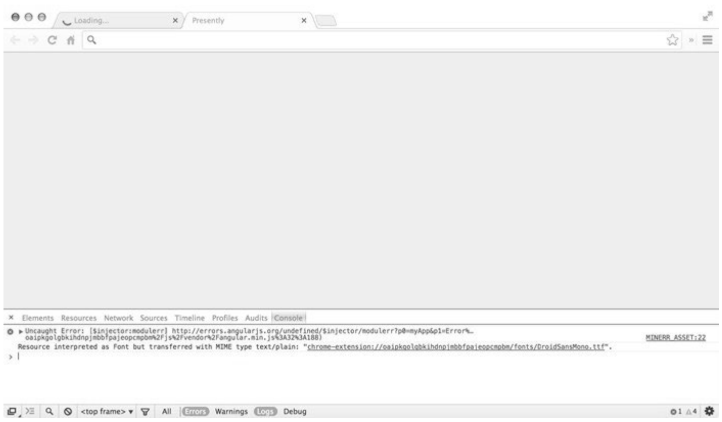


图31-4　在浏览器中加载未打包的扩展

> 无论什么时候，只要更新或者修改了manifest.json文件，都需要在chrome://extensions中点击“重新载入”按钮，重新在幕后连接我们的Chrome应用。

① http://code.angularjs.org/snapshot/angular-csp.css

31.7　主模块

接下来我们将在js/app.js文件中构建这个完整的Angular应用程序。对于生产版本的应用，你可能希望将这些功能分割到多个JavaScript文件中，然后使用Grunt这类工具压缩和合并这些文件。

这个应用叫做myApp，因此要创建一个同名的Angular模块：

```
angular.module('myApp', [])
```

在这一步之后，这个应用运行在浏览器中就不会有任何问题了。

31.8　构建主页

接下来开始构建应用的主页部分。在这部分，将致力于整合能够让应用程序运行起来的组件。在下个部分，将会建立一个多路由的应用程序。

构建时钟

Presently的主要特征是位于应用程序顶端并每秒都更新的大时钟。在Angular中，建立这个时钟相当简单。

从构建负责管理主屏幕的`MainController`开始。在这个`MainController`控制器内，只需设置一个每秒运转一次，同时更新一个局部作用域变量的延时。

```
angular.module('myApp', [])
.controller('MainController', function($scope, $timeout) {
    // 构建date对象
    $scope.date = {};

    // 更新函数
    var updateTime = function() {
        $scope.date.raw = new Date();
        $timeout(updateTime, 1000);
    };

    // 启动更新函数
    updateTime();
});
```

`MainController`内的`updateTime()`函数每秒都会运行，以便更新`$scope.date.raw`时间戳，同时更新视图。

为了看到加载到Chrome应用视图中的所有信息，还需要给文档绑定这个数据。可以使用标准的`{{ }}`模板语法来设置这个绑定：

```
<div class="container">
    <div ng-controller="MainController">
        {{ date.raw }}
    </div>
</div>
```

当返回到浏览器中并刷新它时，应该会看到一个未格式化的日期对象显示在视图中。

目前，浏览器中的这个日期还非常丑陋。我们可以利用Angular内置的过滤器以更优雅的方式来格式化这个日期。

让我们在主屏幕上以类似于普通日期格式的方式格式化这个日期。更新视图时，需要将这个日期移到它嵌套的div中，然后添加显示日期的格式：

```
<div class="container">
    <div ng-controller="MainController">
        <div id="datetime">
            <h1>{{ date.raw | date:'hh mm ss' }}</h1>
        </div>
    </div>
</div>
```

然后借鉴Bootstrap的帮助使用一点CSS，这个日期就会以更人性化的格式显示在屏幕上，如图31-5所示。

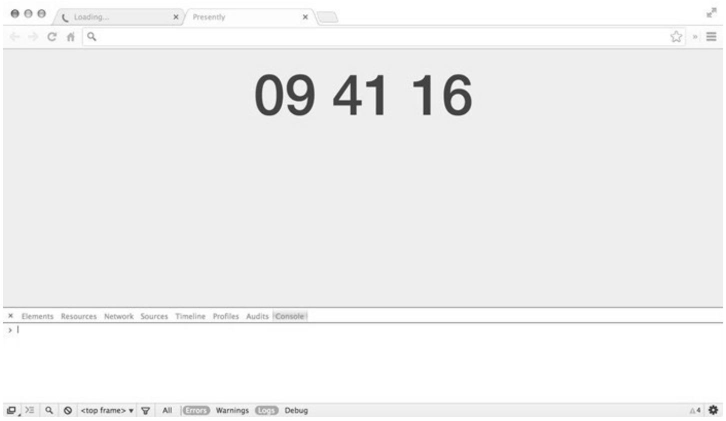


图31-5　第一个界面

这里我们还使用了一点点CSS规则让日期和时间居中显示在屏幕上，同时放大了字体大小以让它突出显示在屏幕上。

```
#datetime {
    text-align: center;
}
#datetime h1 {
    font-size: 6.1em;
}
```

你可以在视图中添加另一个日期，以人性化的方式来显示当前日期。只要再添加一个格式化的日期就可以了：

```
<!-- ... -->
<div id="datetime">
    <h1>{{ date.raw | date: 'hh mm ss' }}</h1>
    <h2>{{ date.raw | date: 'EEEE, MMMM yyyy' }}</h2>
</div>
<!-- ... -->
```

下面的CSS用于给`#datatime h2`简单地增大`<h2>`标签的字体大小，效果如图31-6所示。

```
#datetime h2 {
    font-size: 1.0em;
}
```

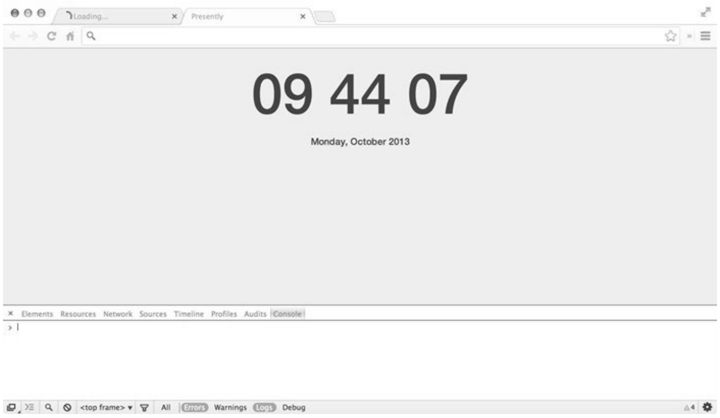


图31-6　完整的日期

31.9　使用 Wundergroud 的天气 API

这个应用还需要使用外部资源来获取我们感兴趣的当前位置的天气信息。在这个应用程序中，我们将使用Wunderground[1]的API。

为了使用Wunderground的API，需要获取一个访问API的密钥。

要获取这个访问API的密钥，首先需要注册。到Wunderground的API页面http://www.wunderground. com/weather/api/点击"Sign Up for free!"即可。

在随后的页面上填好相关信息之后，可以点击提交，直到到达显示API密钥的详情页面。

设置好之后，找出Wunderground的API密钥，然后保存它。我们很快就会用到它。

构建Angular服务

我们不会将逻辑放到控制器中来提取当前天气，因为它效率低下（当你导航另一个页面时，控制器就告吹了，因为每当控制器接入后都需要重新调用这些API），同时把实现细节混入业务逻辑细节中也是不好的设计。

相反，这里应该使用服务。在应用程序生命周期持续的时间里，服务是跨控制器持久存在的，并且这也是避开控制器隐藏业务逻辑的适当位置。

我们需要在应用启动时配置应用，因此需要使用`.provider()`方法创建这个服务。这是创建可以注入到`.config()`函数中服务的唯一方法。

为了构建这个服务，我们将使用`.provider()` API方法，它接受一个服务名以及一个定义实际程序的函数。

① http://www.wunderground.com/

```
angular.module('myApp', [])
.provider('Weather', function() {
})
```

在这个方法内，需要定义一个$get()函数，该函数返回可用于这个服务中的方法。要配置这个服务，还需要一个可以在配置中设置API密钥的方法。而这些方法都应该在$get()函数的作用域外部。

```
.provider('Weather', function() {
    var apiKey = "";

    this.setApiKey = function(key) {
        if(key) this.apiKey = key;
    };

    this.$get = function($http) {
        return {
            // 服务对象
        }
    }
})
```

使用这段小巧的代码，现在，我们可以将Weather服务注入到.config()函数中，然后使用你的Wunderground API密钥来配置这个服务。

当Angular碰到使用.provider() API方法创建的提供者时，它会创建一个可注入对象[Name]Provider。这是一个将会注入到配置函数中的对象：

```
// .provider('Weather', function() {
// ...
// })
.config(function(WeatherProvider) {
    WeatherProvider.setApiKey('YOUR_API_KEY');
})
// .controller('MainController', function($scope, $timeout) {
// ...
```

Wunderground的API要求我们将这个API密钥传递给请求的URL。为了将API密钥传递给每个请求，还需要创建一个生成这个URL的函数。

```
var apiKey = "";
// ...
this.getUrl = function(type, ext) {
    return "http://api.wunderground.com/api/" +
        this.apiKey + "/" + type + "/q/" + ext + '.json';
};
```

现在，我们可以创建API来调用Weather服务，帮我们从Wunderground的API中获取最新的预测数据。

我们将会创建自己的、可以用来在视图中解析数据的promise，因为我们只希望从API调用中返回相关的结果：

```
this.$get = function($q, $http) {
    var self = this;
    return {
        getWeatherForecast: function(city) {
```

31

```
            var d = $q.defer();
            $http({
                method: 'GET',
                url: self.getUrl("forecast", city),
                cache: true
            }).success(function(data) {
                // Wunderground API返回
                //嵌套在forecast.simpleforcast属性内的forecasts对象
                d.resolve(data.forcast.simpleforcast);
            }).error(function(err) {
                d.reject(err);
            });
            return d.promise();
        }
    };
}
```

现在，我们可以将这个Weather服务注入到控制器中，然后在控制器中调用getWeather Forecase()方法以及响应promise，而不是处理API的复杂度。

回到MainController：现在可以注入Weather服务，然后在作用域上设置其结果：

```
.controller('MainController',
function($scope, $timeout, Weather) {
    // ...
    $scope.weather = {};
    // 暂时使用硬编码的San_Francisco
    Weather.getWeatherForecast("CA/San_Francisco")
        .then(function(data) {
            $scope.weather.forcast = data;
        });
        //...
```

要查看视图中调用这个API的结果，需要更新tab.html。出于调试的目的，这里先在一个<pre>标签内使用json过滤器，效果如图31-7所示。

```
<div id="forecast">
    <pre>{{ weather.forecast | json }}</pre>
</div>
```

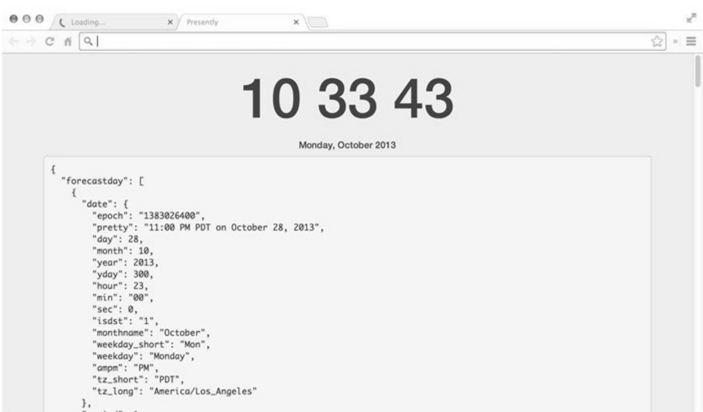


图31-7　调用天气API进行调试

这里可以看到视图已经使用最新的天气数据更新了，现在我们来创建一个更优雅的视图。

视图本身会遍历`forecast.forecastday`集合。对于每个元素，我们会创建一个视图来显示Wunderground API为我们提供的天气图标，以及一个便于阅读的日期和最高温度，如图31-8所示。

```
<div id="forecast">
    <ul class="row list-unstyled">
        <li ng-repeat="day in weather.forcast.forcastday" class="col-md-3">
            <div ng-class="{{ today: $index == 0 }}">
                <img src="{{ day.icon }}" ng-src="{{ day.icon_url }}" />
                <h3>{{ day.high.fahrenheit }}</h3>
                <h4 ng-if="$index == 0">Now</h4>
                <h4 ng-if="$index != 0">{{ day.date.weekday }}</h4>
            </div>
        </li>
    </ul>
</div>
```

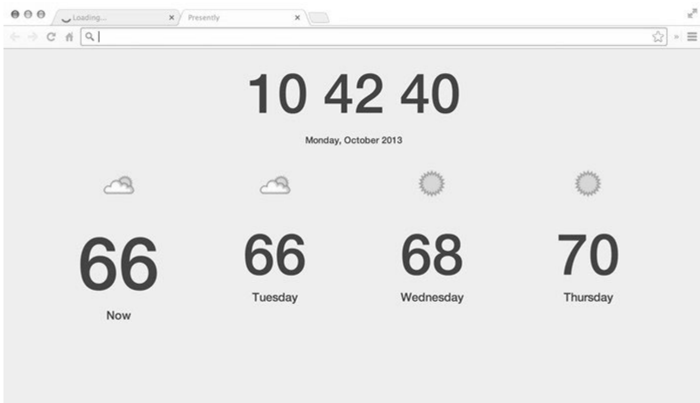


图31-8 干净的HTML天气视图

下面是设置给视图的样式：

```
#forcast ul li {
    font-size: 4.5em;
    text-align: center;
}
#forecast ul li h3 {
    font-size: 1.4em;
}
#forecast ul li .today h3 {
    font-size: 1.8em;
}
```

31.10 设置界面

目前，这个应用还只有一个视图，它带有一个硬编码的城市，用于为每个浏览者提取天气数据。但是这个设置为所有人提供的都是旧金山的数据，因此它还不能为其他地方的人效力。

为了让用户能够使用Presently自定义体验，还需要添加另一个界面：设置界面。

要引进第二个界面（以及多个界面），需要将ngRoute模块作为依赖添加给我们的应用模块。

```
angular.module('myApp', ['ngRoute'])
```

现在，可以定义单独的视图和路由，然后把主界面视图从tab.html主视图中提取出来。

在定义路由期间，注意这里需要两个路由：分别对应应用程序的两个不同界面。

```
// angular.module(...)
// ...
.config(function($routeProvider) {
    $routeProvider
    .when('/', {
        templateUrl: 'templates/home.html',
        controller: 'MainController'
    })
    .when('/settings', {
        templateUrl: 'templates/settings.html',
        controller: 'SettingsController'
    })
    .otherwise({redirectTo: '/'});
})
```

可以将tab.html中`div.container`之间的所有HTML移入templates/home.html文件中，然后使用`<div ng-view></div>`来替换它。

```
<div class="container">
    <div ng-view></div>
</div>
```

刷新页面时，可以看到似乎没有什么变化，但是HTML是从templates/home.html模板中加载的，而不是直接嵌入在tab.html内。

目前，我们还没有切换两个界面的方式。可以添加一些允许用户导航到不同页面的基于页脚导航的元素。在这里，我们仅在页面底部添加两个链接，导航到不同的页面，就像下面这样：

```
<div id="actionbar">
    <ul class="list-inline">
        <li><a class="glyphicon glyphicon-home" href="#/"></a></li>
        <li><a class="glyphicon glyphicon-cog" href="#/settings"></a></li>
    </ul>
</div>
```

为了将它们添加到界面的右下角，必须对它们应用一些绝对定位的CSS：

```
#actionbar {
    position: absolute;
    bottom: 0.5em;
    right: 1.0em;
}
#actionbar a {
    font-size: 2.2rem;
    color: #000;
}
```

现在，如果你通过点击齿轮按钮导航到设置页面，你会看到这里没有呈现任何信息。这需要定义`SettingController`，以便我们可以操作视图，同时供用户使用。

```
// ...
.controller('SettingsController', function($scope) {
    // 这里是控制器定义
});
```

设置界面本身是一个功能单一的表单，负责让用户切换到它们感兴趣的城市。HTML类似于这块代码（其中有一些我们还没实现的功能）：

```
<h2>Settings</h2>
<form ng-submit="save()">
    <input type="text" ng-model="user.location" placeholder="Enter a location" />
    <input class="btn btn-primary" type="submit" value="Save" />
</form>
```

31.11 实现用户服务

出于同样的原因，我们隐藏了Wunderground API的复杂度，而且我们还希望隐藏用户API。这样就可以使用本地存储，以及在应用的任意部分与用户设置控制器通信。

UserService本身非常简单，它不需要在应用中配置。在不使用本地存储的情况下，UserService其实很简单：

```
// ...
.factory('UserService', function() {
    var defaults = {
        location: 'autoip'
    };
    var service = {
        user: defaults
    };

    return service;
})
```

这个服务在应用程序生命周期内都会持有user对象。也就是说，只要浏览器窗口开着，用户对应用程序的设置都会保持不变；然而，如果用户在Chrome中打开一个新标签页，这些设置就会丢失，这是不理想的。

可以通过使用Chrome的sessionStorage功能来跨应用保留用户设置。幸好，这个API非常简单。

只需添加两个函数给UserService：

- save
- restore

即使使用这些功能，UserService的代码也没有增长太多：

```
// ...
.factory('UserService', function() {
    var defaults = {
        location: 'autoip'
    };

    var service = {
        user: {},
        save: function() {
            sessionStorage.presenty = angular.toJson(service.user);
        },
        restore: function() {
```

31

```
            // 从sessionStorage中拉取配置
            service.user = angular.fromJson(sessionStorage.presenty) || defaults
            return service.user;
        }
    };
    // 立即调用它，从session storage中回复配置
    // 因此这里的用户数据是立即可用的
    service.restore();
    return service;
})
// ...
```

我们还可以跨Chrome应用注入这个UserService，然后使用同样的用户数据。回到SettingsController，现在可以使用这个新服务设置user对象来定义设置信息：

```
.controller('SettingsController', function($scipe, UserService) {
    $scope.user = UserService.user;
});
```

如果刷新浏览器，会看到有一个为用户设置的默认autoip，这是我们在UserService中定义设置的默认值。

我们需要一种将用户数据保存在会话存储中的方法，这样就可以跨应用使用这些数据了。在templates/settings.html中，定义了一个带有ng-submit="save()"行为的表单；因此，当用户提交这个表单时，save()函数就会被调用。

在SettingsController内，我们需要实现这个save()函数，它会调用UserService上的save方法，将用户数据保存到他们的sessionStorage中。

```
.controller('SettingsController', function($scope, UserService) {
    $scope.user = UserService.user;
    $scope.save = function() {
        UserService.save();
    }
});
```

还有一个绑定到user.location的唯一输入字段，如果用户改变它的值并按下保存，用户的sessionStorage就会更新，如图31-9所示。

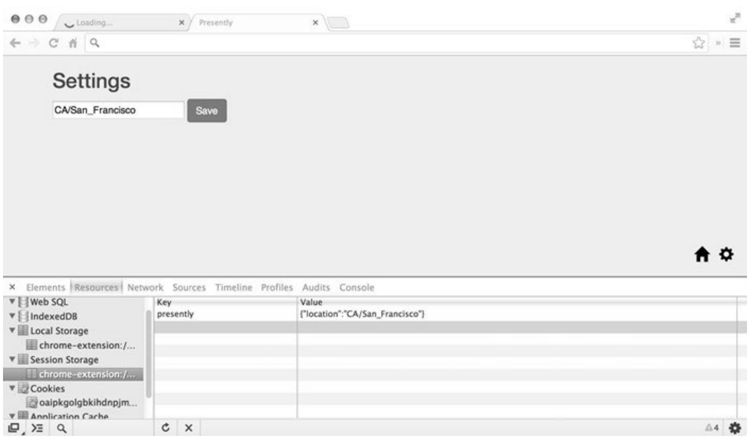


图31-9　sessionStorage

通过在HomeController中使用UserService，现在我们可以移除硬编码的值“CA/San_Francisco”，然后使用新的UserService对象返回的地址替换它。

```
// ...
.controller('MainController', function($scope, $timeout, Weather, UserService) {
    // ...
    $scope.user = UserService.user;
    Weather.getWeatherForecast($scope.user.location).then(function(data) {
        $scope.weather.forecast = data;
    });
    // ...
})
```

正如我们可以看到的那样，如果切换到设置视图，然后输入“NY/New_York”，可以看到天气会根据我们在设置页面中设置的地址发生改变，如图31-10所示。

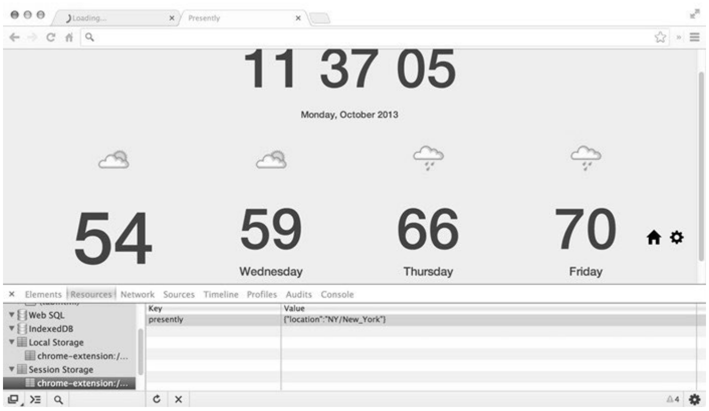


图31-10　纽约天气

31.12　城市自动填充/自动完成

每次都需要输入一个符合Wunderground API风格的城市很不方便（经度/维度、城市和州、国家代码，等等）。幸好，Wunderground API还为我们提供了一个autocomplete API[①]。

无需要求用户知道具体城市的格式，我们可以提供一个列表，供用户选择。

为了保持简单性和灵活性，这里只会创建一个未加工的基于JavaScript的自动完成，而不是使用插件，比如typeahead.js或者jQuery插件库。

要做到这一点，我们要在<input>元素上创建一个指令，用来插入一个带建议地址的<ul>元素。

```
.directive('autoFill', function($timeout) {
    return {
        restrict: 'EA',
        scope: {
            autoFill: '&',
```

① http://www.wunderground.com/weather/api/d/docs?d=autocomplete-api

```
            ngModel: '='
        },
        compile: function(tEle, tAttrs) {
            //编译函数
            return function(scope, ele, attrs, ctrl) {
                //链接函数
            }
        }
    }
});
```

因为这里会创建一个新元素，因此需要使用compile函数而不只是link函数和模板，因为<ul>元素不能嵌套在一个<input>元素中。

这里不会深入介绍compile函数的工作原理，只会创建一个新元素，然后为它设置绑定：

```
// ...
compile: function(tEle, tAttrs) {
    var tplEl = angular.element('<div class="typeahead">' +
    '<input type="text" autocomplete="off" />' +
    '<ul id="autolist" ng-show="reslist">' +
      '<li ng-repeat="res in reslist" ' +
      '>{{res.name}}</li>' +
    '</ul>' +
    '</div>');

    var input = tplEl.find('input');
    input.attr('type', tAttrs.type);
    input.attr('ng-model', tAttrs.ngModel);
    tEle.replaceWith(tplEl);

    return function(scope, ele, attrs, ctrl) {
        // ...
```

在链接函数内，我们需要给keyup事件绑定一个函数，同时还要检查在输入字段中至少有最少数量的字符。只要发现有最少数量的字符，就运行通过使用指令设置的函数，来提取自动建议值。

Autocomplete API

让我们来看看如何调用这个指令：通过传递一个函数调用给auto-fill指令，然后将位置值绑定给user.location。

```
<input type="text"
    ng-model="user.location"
    auto-fill="fetchCities"
    autocomplete="off"
    placeholder="Location" />
```

在Weather服务中，我们会创建另一个函数明确地调用autocomplete API，并解析一个带有与查询词汇对应的完整建议列表的promise。

```
getWeatherForecast: function(city) {
    // ...
},
getCityDetails: function(query) {
    var d = $q.defer();
    $http({
```

```
        method: 'GET',
        url: "http://autocomplete.wunderground.com/aq?query=" + query
    }).success(function(data) {
        d.resolve(data.RESULTS);
    }).error(function(err) {
        d.reject(err);
    });
    return d.promise;
}
```

回到SettingsController中，我们可以引用这个函数来检索建议值列表。记住，需要在控制器中注入这个Weather服务来引用它。

```
.controller('SettingsController',
  function($scope, UserService, Weather) {
     // ...
     $scope.fetchCities = Weather.getCityDetails;
    });
```

在这个指令中，可以调用这个函数，我们想在这个函数引用的动作上触发修改。

```
// ...
tEle.replaceWith(tplEl);
return function(scope, ele, attrs, ctrl) {
    var minKeyCount = attrs.minKeyCount || 3,
        timer,
        input = ele.find('input');

    input.bind('keyup', function(e) {
        val = ele.val();
        if(val.length < minKeyCount) {
            if(timer) $timeout.cancel(timer);
            scope.reslist = null;
            return;
        } else {
            if(timer) $timeout.cancel(timer);
            timer = $timeout(function() {
                scope.autoFill()(val).then(function(data) {
                    if(data && data.length > 0) {
                        scope.reslist = data;
                        scope.ngModel = data[0].zmw;
                    }
                });
            }, 300);
        }
    });

    // 失去焦点时隐藏reslist
    input.bind('blur', function(e) {
        scope.reslist = null;
        scope.$digest();
    });
}
```

这里用了一个延时，以便只在用户输入完成后调用该函数。使用延时只是一种阻止函数被重复调用的简单方式，而我们真正感兴趣的只是第一次调用suggestion API，如图31-11所示。

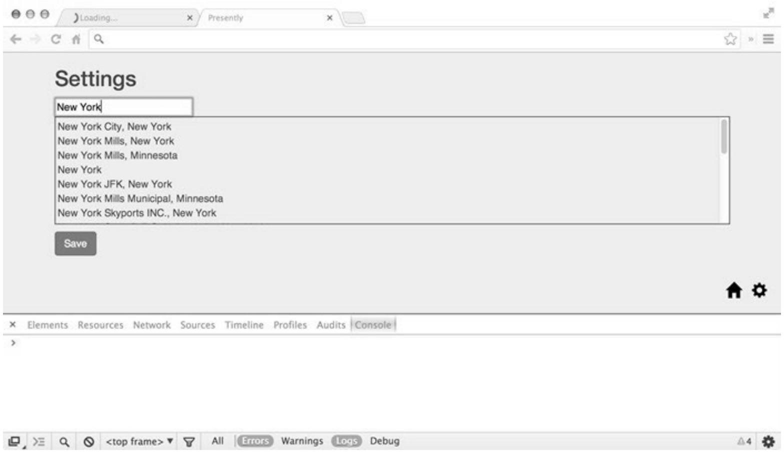


图31-11　在浏览器中自动填充

31.13　添加时区支持

最后，我们还希望能够根据用户在他的设置中的选择来更新时钟和显示新地址。要更新时钟包含时区支持很容易：实际上我们已经通过自动完成API实现了最难的部分。

首先，必须给指令再添加一个用作时区的timezone属性：

```
<input type="text" ng-model="user.location" timezone="user.timezone" auto-fill="fetchCities"
    autocomplete="off" placeholder="Location" />
```

接下来，需要在指令的compile函数中将timezone属性添加给生成的<input>字段：

```
// ...
input.attr('type', tAttrs.type);
input.attr('ng-model', tAttrs.ngModel);
input.attr('timezone', tAttrs.timezone);
tEle.replaceWith(tplEl);
// ...
```

最后一项要点，在处理自动完成的链接函数中，还要在保存用户位置值时保存用户的时区：

```
// ...
scope.reslist = data;
scope.ngModel = data[0].zmw;
scope.timezone = data[0].tz;
// ...
```

回到浏览器中，当用户输入城市时，会看到我们在新的城市值旁边保存了时区信息，如图31-12所示。

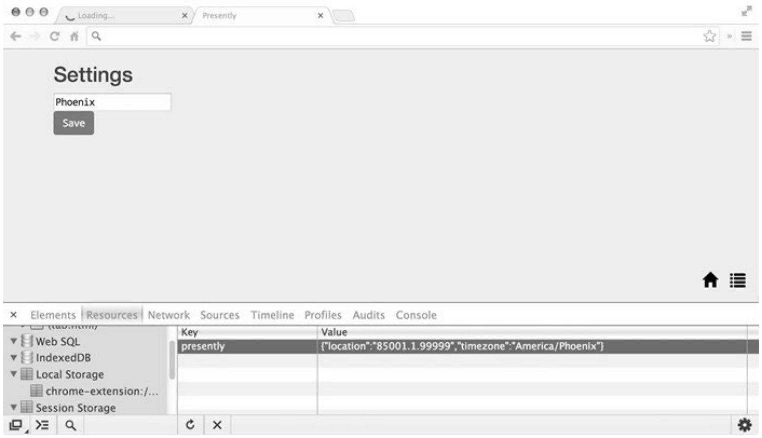


图31-12 时区支持

至此，我们还需要更新MainController内的时间和日期，把新增的时区考虑进去。

以前，匹配时区名对应的格林尼治时间（GMT）偏移量是一项艰巨的任务。Mozilla和Chrome团队使用新的timeZone参数实现了toLocaleString方法，让我们就能够根据时区信息重新映射日期。由于这里编写的是一个Chrome应用，因此可以在应用中大胆地使用这个函数。

回到MainController中，我们还可以基于之前保存的时区信息创建一个新日期：

```
.controller('MainController', function($scope, $timeout, Weather, UserService) {
    $scope.date = {};

    var updateTime = function() {
        $scope.date.tz = new Date(new Date().toLocaleString("en-US", {timeZone: $scope.user.
timezone}));
        $timeout(updateTime, 1000);
    }
    // ...
```

现在，我们不再在视图中使用$scope.date.raw了，而是使用$scope.date.tz。时间会伴随时区的修改而变化，如图31-13和图31-14所示。

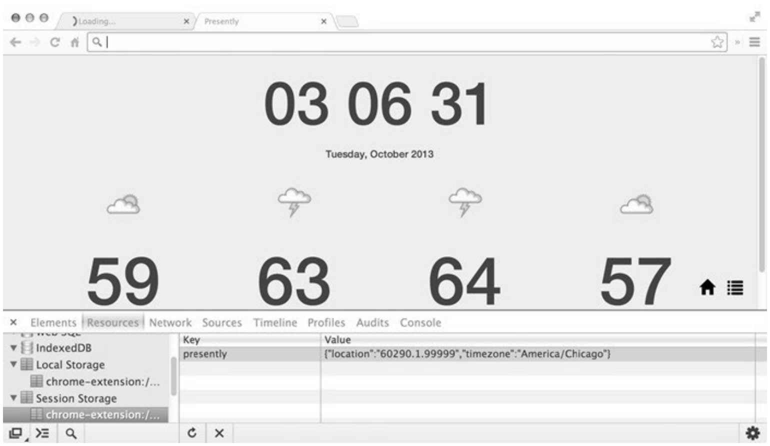


图31-13 芝加哥时区

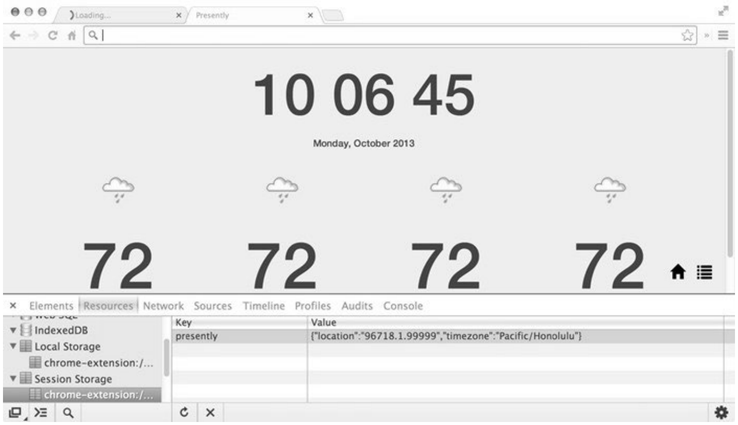

10 06 45
Monday, October 2013
72
72
72
72
Elements
Resources
Network
Sources
Timeline
Profiles
Audits
Console
IndexedDB
Local Storage
chrome-extension:/...
Session Storage
chrome-extension:/...
Key
Value
presently
{"location":"96718.1.99999","timezone":"Pacific/Honolulu"}

图31-14　夏威夷时区

第 32 章

优化Angular应用

32

显然，Angular在开发期间就做了很大的优化，但是在性能方面它是如何定位的呢？为了让它尽可能快，我们还能做些什么呢？

对于大多数Web应用程序而言，Angular的速度已经足够快了，因此无需特意优化它的性能。当你的应用程序变慢或者表现不佳时，才需要动手优化Angular应用。

32.1 优化什么

为了了解应该集中优化应用的哪些部分，需要理解Angular在幕后发生了什么事情。对于任何应用程序，我们都应该从问题的起因开始。

32.2 优化$digest循环

查找性能问题，最明显的地方是从$digest循环开始。简而言之，Angular通过运行一个监控列表来跟踪实时数据绑定。页面上每一个可能改变的实时数据都有一个监控函数。

> 第24章详细讨论了$watch列表和$digest循环。

每个监控列表都可能导致$digest循环花更多的时间完成渲染，因为Angular需要跟踪值，同时检查它在每个循环中是否发生变化。

限制不必要的监控数量会获得最大的性能提升。此外，双向数据比较应尽量简单，这样会带来更多的性能提升，因为这样浏览器可以快速地检查它们。

在你的应用中，你应该留心双向数据绑定的数量，对于页面上每个$digest循环中的数据绑定不应该超过2000个。

有时，甚至不需要在应用程序中运行完整的$digest循环。想象一下，有个一秒钟查询服务器多次的轮询。如果我们收到一个websocket事件，触发一个完整的digest循环来处理每条消息，这将会得到相当慢的应用程序。

> 推荐使用websocket，因为它们在生产中更加友好并且不容易出错。

```
app.factory('poller', function($rootScope, $http) {
    var pollForEvent = function(timeout) {
        $http.get('/events')
        .success(function(data) {
```

```
            var events = data.events;
            for(var i = 0;l i < events.length; i++) {
                var event = events[i];
                if(servie.handlers[event])
                    for(handler in service.handlers[event])
                        $rootScope.$apply(function() {
                            handler.apply(event);
                        });
            }
            // 设置下一个延时
            setTimeout(pollForEvent, timeout);
        });
    };
    // 每半秒轮询一次
    setTimeout(function() { pollForEvent(500); });
    var service = {
        handlers: {},
        on: function(evt, callback) {
            if(!service.handlers[evt])
                service.handlers[evt] = [];

            service.handlers[evt].push(callback);
        }
    }
    return service;
});
```

这段代码的主要问题在于，每个独立事件被触发时就会结束运行`$rootScope.$apply()`方法，这会导致每秒产生很多`$digest`循环。

限制每秒钟`$digest`循环的数量是提升应用性能的一个很好方式。你可以使用事件节流[①]的方式设置一个每秒触发事件的最大次数。

```
// 节流函数
var throttle = function(fn, atMost, ctx) {
    var ctx = ctx || this;
    var atMost = atMost || 250; // 毫秒
    var last, defer, result;
    return function() {
        var now = new Date(),
            args = arguments;
        if (last && now < last + atMost) {
            // 延迟执行
            clearTimeout(defer);
            defer = setTimeout(function() {
                last = now;
                fn.apply(ctx, args);
            }, atMost);
        } else {
            result = fn.apply(ctx, args);
        }
        return result;
    }
}
```

在每个`atMost`生命周期内，这个比较丑陋的`throttle()`函数最多只会触发函数一次。

① 节流，一种处理连续重复执行的操作的策略。旨在限制无间断地重复执行函数或者事件。——译者注

这个代码在Underscore.js[1]库中有一个可用于生产并经过良好测试的更优版本。

为了使用throttle函数设置$digest循环节流，你可以在事件循环中调用它：

```
// ...
for(var i = 0; i < events.length; i++) {
    var event = events[i];
    if(service.handlers[event])
        for(handler in service.handlers[event])
            throttle(function() {
                $rootScope.$apply(function() {
                    handler.apply(event);
                });
            }, 500);
}
```

32.3　优化 ng-repeat

Angular中最大的延迟资源之一便是ng-repeat指令。对于每个ng-repeat所处的独立元素，列表中每个条目都至少有一个数据绑定，这甚至还没有包括创建在列表元素内的任何其他绑定。

让我们来看看下面这个使用ng-repeat生成的重复列表的性能：

```
<ul>
    <li ng-repeat="email in emails">
        <a ng-href="#/from/{{ email.sender }}">
            {{ email.sender }}
        </a>
        <a ng-href="#/email/{{ email.id }}">
            {{ email.subject }}
        </a>
    </li>
</ul>
```

对于这个列表中每个独立的email，最少都有一个由ngRepeat指令生成的监控函数（这个监控函数用于监控列表的变化）。由于Angular会为每个独立的ng指令创建一个$watch，上面的列表中每个email都会有4+1个监控器。对于一个有100个email的列表，上面的用法就会创建500个监控器，而上面这个例子甚至都不是一个复杂的完整页面。

对于这个相对简短的列表，很明显它会大大降低一个大型应用程序的性能。这里有一些相对简单的方式可以用来加速你的应用程序。

32.4　优化$digest 调用

在改变一个变量时通常可以确定什么时候会运行$disgest循环，以及运行$digest循环会影响哪些作用域。在这种情况下，你无需在$rootScope上使用$scope.$apply()（这会导致每个子作用域$scope跑进$digest循环中）调用完整的$digest循环。作为替代可以直接调用$scope.$digest()。

① http://underscorejs.org/

调用$scope.$digest()只会在调用了$digest()及其子节点的具体作用域上运行digest循环。

32.5　优化$watch 函数

由于$watch列表中的表达式会在每个$digest循环中执行，保持较少的功能很重要。更小以及更专注的$watch表达式，会让应用程序的性能更好。

在$watch()函数中避免深度比较、复杂的逻辑以及使用少量的循环，会有助于加速应用程序。

例如，可以设置一个监控函数来监控一个对象。假设有一个Account对象：

```
$scope.account = {
    active: true,
    userId: 123,
    balance: 1000 // 美分
}
```

假定想要监控任意时刻的账户余额变化，然后在余额为0时设置账户为未激活。可以设置一个$watch函数来监控这个account对象，每当余额对象变化时更新账户信息：

```
$scope.$watch('user', function(newAccount) {
    if(newAccount.balance <= 0) {
        $scope.account.active = false
    }
}, true);
```

$watch()函数中的第三个参数用于告诉Angular，是否使用深度比较来监控这个对象，它会使用angular.equals()函数检查每个属性。

这个选择将会导致糟糕的性能。不仅Angular会创建一个对象副本，在存储它时需要遍历每个属性来检查其中是否有任何变化。

这里有一个构建$watch函数的技巧：使用它们跟踪明显会影响视图的变量。对于不会影响视图的任何事物都不需要使用$watch函数。

有时候移除监控器对我们来说是有意义的，特别是在数据是静态的，并且只想在第一时间将它暴露给视图的时候，因为这时$watch函数就变得无关紧要了。

从视图中移除自定义的监控器也很容易：让$watch函数自身返回一个为我们移除$watch的函数即可。

例如，比方说有一个自定义的指令等待解析变量name：

```
<div data-my-directive name="customerName"></div>
```

由于这个customerName一旦设置后就不可改变，你可以通过移除已经设置的$watch函数来优化这个指令：

```
.directive('myDirective', function($q) {
    return {
        // ...
        scope: {
            name: '='
        },
```

```
        link: function(scope, ele, attrs, ctrl) {
            var unWatch = $scope.$watch(attrs.name, function(n, o) {
                if(n != o) {
                    // 使用解析后的name做些什么事
                    // 然后移除watch
                    unWatch();
                }
            });
        }
    };
});
```

可以通过移除应用程序中所有不必要的监控器来提升性能。当你尝试从应用程序中移除每个监控函数时，这个过程可能特别麻烦，特别是当你尝试移除由Angular设置的默认监控器时。

还可以编写自定义的指令来维护监控器，而不是使用Angular提供的内置指令。幸好，你不必自己编写这些指令，因为有一个叫做bindonce的库已经为我们编写好了。

32.5.1 bindonce

bindonce是一个可以用于你的应用中的即插即用模块，它只保留了监控一次的指令；它还为我们提供了传递异步数据的能力。

这个库提供了新的指令，用于不需要实时更新的DOM元素。这些指令会留意值的填充和验证。一旦数据可用，它就渲染它以及子元素的内容，然后立即移除监控器。

使用bindonce指令时创建的独立的临时监控器会在数据变得可用时被移除。如果数据在作用域中已经可用了，它不会创建监控器，而是渲染子元素。

回想一下上一个例子。我们将使用bindonce的非永久监控器（zero permanent watcher）创建同样的例子：

```
<ul>
    <li bindonce="email" ng-repeat="email in emails">
        <a bo-href-i="#/from/{{ email.sender }}" bo-text="email.sender"></a>
        <a bo-href-i="#/email/{{ email.id }}" bo-text="email.subject"></a>
    </li>
</ul>
```

要使用bindonce，首先要获取其源码。可以直接从Github上的项目页面（https://github.com/pasvaz/bindonce）得到，或者像这样使用Bower安装它：

```
bower install angular-bindonce
```

拿到源码之后，需要在主视图中引用它：

```
<script src="scripts/vendor/bindonce.js"></script>
```

最后，还需要将它作为一个依赖设置给应用程序模块：

```
angular.module('myApp', ['pasvaz.bindonce']);
```

现在，当我们处理静态数据时，我可以保证，在bo-*标签的帮助下，我们无需使用不必要的监控器了。

bindonce库为我们提供了很多指令。正如上面看到的，我们使用了两个自定义指令。

> 使用bo-*标签时，需要确保包含了bindonce指令。所有子bo-*指令都会等待这个指令解析数据。

1. bo-if="condition"

这个指令等同于调用ng-if，但是它没有使用额外的监控器。

2. bo-show="condition" / bo-hide="condition"

这个指令等同于调用ng-show或者ng-hide，但是没有使用任何额外的监控器。

3. bo-text="text"

这个指令会对text求值，然后将它放到元素内。类似于ng-bind。

4. bo-href="url" / bo-href-i="url"

使用bo-href时不允许使用需要插值的"url"，而bo-href-i允许URL中包含插值。下面这两个调用功能是等价的：

```
// bo-href 不允许任何插值
<a bo-href="'/users/' + User.id">√</a>
// bo-href-i 允许插值
<a bo-href-i="'/users/{{ User.id }>√</a>
```

5. bo-src="url" / bo-src-i="url"

bo-src不允许在URL内插值，而bo-src-i允许。下面这两个调用功能是等价的：

```
// bo-src不允许任何插值
<img bo-src="'/users/' + User.gravatar" />
// bo-src-i允许插值
<img bo-src="/users/{{ User.gravatar }}" />
```

6. bo-alt="text"

类似于bo-text，这个指令会在DOM元素内渲染文本，然后将文本设置给元素的alt属性。

7. bo-title="title"

bo-title指令会在DOM元素内渲染文本，然后将文本设置给元素的title属性。

8. bo-id="id"

这个指令渲染"id"，然后将这个id设置给元素的id属性。

9. bo-style="style"

这个指令会使用和ng-style一样的语法将样式作为表达式渲染，而不会使用监控器。

10. bo-value="value"

这个指令渲染给定的值，然后将它设置给元素的value属性。

11. bo-attr bo-attr-foo="hello"

这个指令会在DOM元素中将文本"foo"作为自定义属性渲染。

使用ng-repeat优化静态数据页面，Bindonce是一个很不错的选择。

32.5.2　$watch函数的自动优化

最新版的Angular在找到恒定值时（比如，表达式解析为布尔值或者静态的整数）会自动移除$watch函数。

```
// 下面这些监控器会被自动移除
// 因为Angular的$watches会检测到这些值都是不会改变的值
$scope.$watch('true', function() {});
$scope.$watch('2 + 2', function() {});
```

32.6　优化过滤器

位于视图中的每个过滤器将至少被调用2次，这是过滤器的本质。越是保持这些函数轻量以及对它们进行优化，应用程序就会更快。

32.6.1　不变的数据

出于这个原因，分析在哪里以及为什么在视图中使用过滤器通常是个好注意。一个好的经验法则：任何可以从视图中移除的过滤器都需要改造一下。

也就是说，当检索到数据时，可以转换这个模型，而不是在视图中使用currency过滤器。那样，我们就不必在视图中运行两次过滤器。

也就是说不要在视图中使用过滤器，就像这样：

```
<!-- 使用过滤器 -->
<div>{{ receipt.total_cost | currency }}</div>
```

可以使用$filter服务在你的控制器（或者服务）中转换这个receipt.total_cost。

```
.controller('ReceiptController', function($scope, $filter) {
    $scope.receipt_total_sum = 12345;
    $scope.receipt.total_cost = $filter('currency')($scope.receipt.total_sum);
});
```

32.6.2　过滤后的数据

我们还有其他的过滤器，比如实时搜索过滤器，它会使用ng-repeat以及通过使用用于分类的orderBy过滤器限定集合中的重复数据。在这些情况下，数据并不会改变，只是设置数据如何显示在屏幕上。

我们可以缓存分类过的、筛选过的结果，只在必要时执行排序，而不是在每个$digest循环中调用这些过滤器。这个变量缓存叫做记忆缓存。

记忆缓存[①]是一种用于加速应用程序的优化技巧，用于这些情况：先前已经调用过函数，对于给定输入，函数结果没有像预期那样发生改变。

① Memoization是一种加速程序的优化技术，为了防止重复计算，它会保留程序的调用结果。更多信息参考维基百科：Memoization（en.wikipedia.org/wiki/Memoization）。——译者注

为了使用记忆缓存替代过滤器，需要实现自定义的memoize()函数或者使用第三方库提供的函数，比如Underscore.js[①]或者Lo-Dash[②]库，这些库都包含了自定义的实现。由于这个函数本身很小，在这里我们引入了Lo-Dash库中的这个基本函数定义：

```
function memoize(fn, resolver) {
    var memoized = {
        var cache = memoized.cache,
            key = resolver ? resolver.apply(this, arguments) : +new Date() + '' + arguments[0];
        return hasOwnProperty.call(cache, key) ? cache[key] : (cache[key] = fn.apply(this,
arguments));
    }
    memoized.cache = {};
    return memoized;
}
```

本质上，这个函数接受两个参数：一个要缓存的函数和一个可选的resolver函数，它用来为存储结果确定缓存键。这个函数返回一个备忘函数。

使用它时，可以将这个备忘函数设置为作用域对象的函数调用，这样就可以在视图中调用它了。

```
angular.module('myApp', [])
.controller('MainController', function($scope $filter) {
    $scope.getNames = memoize(function() {
        return $filter('orderBy')(
            $scope.names,
            $scope.orderBy,
            $scope.reverseList
        );
    },
    function() {
        // resolver函数返回一个表示缓存键的字符串
        return $scope.orderBy + '-' + $scope.reverseList;
    });
});
```

当在视图中调用getNames()时，第一次它会调用orderBy过滤器。第二次调用过滤器时，它并不会运行，因为此时缓存中已经包含了排序后的键。

有时，你可能希望手动清除缓存（比如，添加或者移除条目更新数据）。由于缓存就在函数对象上（作为函数的属性存在），因此可以通过设置它的值为一个新的{ }对象简单地清除它。

```
$scope.getNames.cache = {};
```

32.7　页面加载优化技巧

我们还可以优化在客户端浏览器中渲染页面时需要花的时间。当然，这里并没有银弹[③]可以确定客户端加载页面的最佳机制，正如大部分问题都取决于服务端的组成部分、位置和主机。

① http://underscorejs.org/

② http://lodash.com/

③ 没有银弹，软件工程中对软件工程问题的一种描述。更多信息参考维基百科：没有银弹（http:t.cn/zWIKVjT）。
——译者注

32.7.1 压缩

压缩代码是优化可感知的页面加载时间最容易的方法。

压缩就是从源代码中移除所有不必要的字符，将变量减小到我们可以获取的最小尺寸，撤销注释和块分割符等操作的一个过程。

这就减少了文件进行网络传输所花的时间，因为最终的文件大小减小了。

我们可以压缩HTML、JavaScript、CSS，甚至是图片。压缩并不会牺牲应用程序的功能，反而用户会得到更好的体验。

有许多免费的工具可用于处理压缩。推荐使用`uglify`，可以通过Grunt使用它。关于Grunt的详细信息，请参考34.3节。

32.7.2 利用`$templateCache`

在生产中部署应用时，我们都希望应用的加载尽可能快，以及尽可能做出响应。使用XHR加载模板可能会导致Web应用缓慢或者有卡顿的感觉。可以通过将模板包装为JavaScript文件，然后连同应用程序的其他部分一起传输的方式伪造模板缓存加载，而不是通过XHR提取模板。

关于如何有效地包装模板的详细信息，请参考`$templateCache`工具：`grunt-angular-templates`。

第 33 章 调试AngularJS

当我们构建大型Angular应用时，经常会遇到一些很难发现和解决的问题，从而让人抓狂。

33.1 从 DOM 中调试

尽管这既不是必要步骤，也不一定要从这个步骤开始，但我们可以访问附加给任意DOM元素的Angular属性。我们可以使用这些属性来窥探数据是如何流入应用程序的。

> 永远都不应该依靠还在应用程序生命周期内的DOM元素来获取该元素的属性。这项技术一般都是出于调试的目的才使用的。

为了从DOM中获取这些属性，我们需要找到感兴趣的元素。如果有完整的jQuery库可用，可以使用jQuery的选择器语法：`$("selector")`。

然而，我们并不需要依赖于jQuery从DOM中获取目标元素。相反，我们可以使用`document.querySelector()`方法。

> 注意，`document.querySelector()`并不是在所有浏览器中都可用，它一般适合选择不复杂的元素，而Sizzle[①]（jQuery也使用这个库）或者jQuery[②]支持更复杂的选择。

可以通过选择`ngApp`指令所在的元素来从DOM中检索`$rootScope`，然后将它包装为一个Angular元素（使用`angular.element()`方法）。

对于一个Angular元素，可以从DOM内部调用不同的方法来检查我们的Angular应用。这样做，需要从DOM中选择元素。在只使用JavaScript和Angular的情况下（这里的意思是除了使用Angular，不使用其他任何库），可以以下面这种方式实现：

```
var rootEle = document.querySelector("html");
var ele = angular.element(rootEle);
```

可以使用这个元素提取应用程序的不同部分。

① http://sizzlejs.com/

② http://jquery.com/

33.1.1 scope()

在元素上使用scope()方法时，可以从该元素（或者父元素）上提取它的$scope对象：

```
var scope = ele.scope();
```

使用元素的作用域时，可以检查任意作用域属性，比如在控制器中设置给作用域对象的自定义变量。还可以窥探元素，查看它的$id、$parent对象、设置给它的$$watchers，甚至是手动遍历它的作用域链。

33.1.2 controller()

通过使用controller()方法可以提取当前元素（或者父元素）的控制器：

```
var ctrl = ele.controller();
// 或者
var ctrl = ele.controller('ngModel');
```

33.1.3 injector()

通过在被选中的元素上使用injector()方法可以提取当前元素（或者包含它的元素）的注入器。

```
var injector = ele.injector();
```

然后可以使用这个注入器在应用内实例化任意Angular对象，比如服务、其他控制器或者任意其他对象。

33.1.4 inheritedData()

通过在元素上使用inheritedData()方法可以提取与该元素$scope对象关联的数据。

```
ele.inheritedData();
```

这个inheritedData()方法就是Angular在作用域链中查找数据的方式，它会遍历DOM直到找到一个特定的值或者直到找到最顶层的作用域。

> 如果你使用Chrome，可以使用开发者工具提供的一些快捷方式。比如要简单地查找你所感兴趣的元素，只需在浏览器中右击，然后选择审查元素。这个元素本身存储在一个叫做$0的变量中，然后你可以通过调用angular.element($0)的方式提取被Angular化的元素。

33.2 调试器

Google的Chrome[①]浏览器有自己的调试器工具，它可以在我们的代码中创建断点。使用debugger语句会导致浏览器冻结代码执行，这就允许我们检查真实应用程序内正在运行的代码，

① https://www.google.com/chrome

以及接近在浏览器内执行的代码。

要使用debugger，只需将它添加到应用程序代码的上下文内即可：

```
angular.module('myApp')
.factory('SessionService', function($q, $http) {
    var service = {
        user_id: null,
        getCurrentUser: function() {
            debugger; // 在这个函数内设置debugger
            return service.user_id;
        }
}

        return service;
    });
```

在这个服务内，我们将会调用debugger方法有效地冻结应用程序。

只要浏览器中的Chrome开发工具开着，还可以在这个应用程序代码执行的位置使用console.log()或者其他JavaScript命令。

当完成应用程序代码调试时，需要确保移除这行代码，因为它会冻结浏览器，甚至是产品。

33.3　Angular Batarang

Angular Batarang是一个由Google的Angular团队开发的Chrome扩展程序，它很好地集成了调试Angular应用的工具，如图33-1所示。

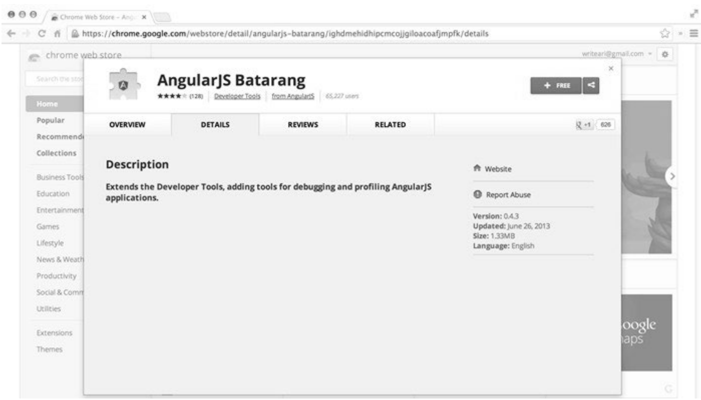


图33-1　Batarang Chrome扩展

33.3.1　安装 Batarang

要安装Batarang，需要从Web Store或者Github仓库https://github.com/angular/angularjs-batarang中下载这个应用程序。

安装好之后，就可以导航到开发者工具中启动这个扩展程序，然后点击enable来启用Batarang收集页面的调试信息。

Batarang允许我们查看Angular应用的作用域、性能、依赖和其他关键参数。

33.3.2 检查模型

启动Batarang之后，页面会重新加载，然后你会注意到有一个面板允许我们选择页面中的不同作用域。

通过点击+按钮可以选择一个作用域，找到你所感兴趣的元素，然后点击它即可。

一旦使用这个检查器选择了一个作用域，就可以查看该作用域元素上的所有属性以及它们的当前值，如图33-2所示。

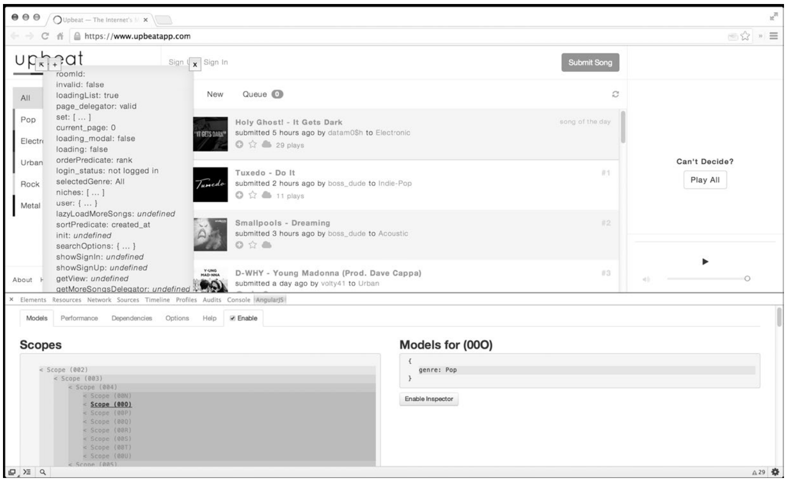


图33-2 模型检查器

33.3.3 检查性能

还可以通过使用Batarang的性能部分来检测应用程序的性能。

在下面这个面板中，可以看到应用程序不同作用域的监控列表，以及计算每个表达式所花的时间，既有真实的时间，也有总体应用程序执行时间的百分比，如图33-3所示。

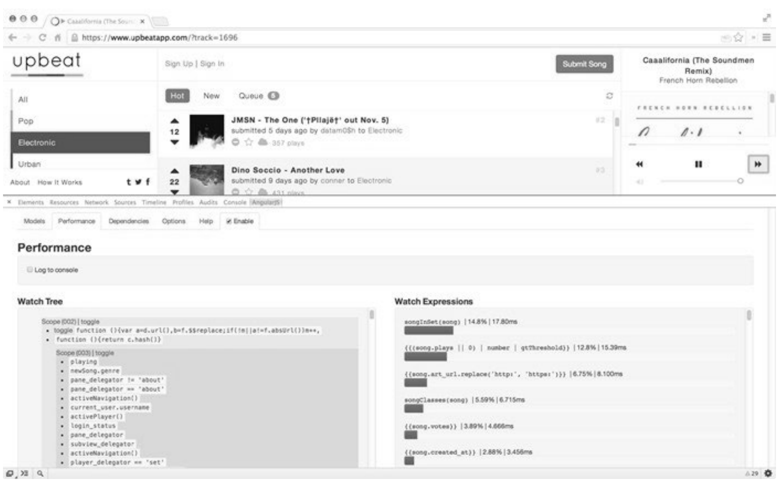


图33-3 性能检查器

33.3.4　检查依赖图表

Batarang工具的一个非常不错的特性就是提供了一个内联的可视化依赖图表。在这里，可以查看应用程序的依赖，查看应用程序所依赖的不同的库，以及跟踪应用程序并不完全依赖的库，如图33-4所示。

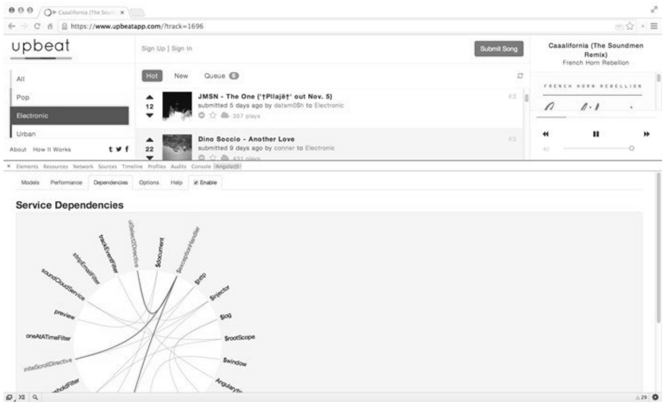


图33-4　依赖图表

33.3.5　可视化应用

Batarang还允许我们在页面上深入查看应用程序。使用选项面板时，我们可以看到：

Applications 独立页面上的不同应用程序（使用的ngApp指令）。

Bindings 设置在视图中的绑定，使用的`ng-bind`的位置或者包裹在模板标记`{{ }}`中的元素。

Scopes 视图中的目标作用域，并且可以深入检查。

选项面板还允许我们查看应用程序所用的Angular版本，以及从CDN中使用了什么，没使用什么，如图33-5所示。

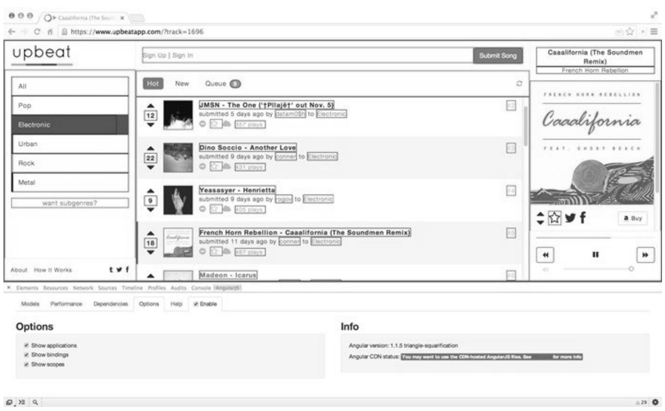


图33-5　选项面板

总而言之，当我们要深入了解Angular应用是如何实时工作时，Batarang工具为我们提供了很多便利。

第 34 章

下一步

34

现在，我们已经熟悉AngularJS了，接下来一起看看那些可用于生产环境的专业工具。

34.1 jqLite 和 jQuery

尽管Angular不鼓励依赖jQuery库，但是在应用中仍然可以使用它。只需确保在DOMContentLoaded事件被触发之前载入它或者手动启动应用即可。

Angular本身包含了一个叫做jqLite的可兼容性库。

本书使用过的angular.element()方法就返回一个jqLite对象，jqLite是jQuery库的子集，它允许Angular以跨浏览器兼容的方式维护DOM。

jqLite并不试图覆盖jQuery库包含的所有方法，它旨在保持轻量，并且只覆盖了Angular要用到的那些方法。

这个库包含以下jQuery方法。

addClass() 给元素添加指定的类。

after() 在元素的后面插入内容。

append() 将内容插入到元素的尾部。

attr() 获取或者设置元素的属性[①]值。

bind()/on() 给选中元素的一个或者多个事件附加一个事件处理程序。

children() 获取元素的子元素。

clone() 创建一个元素的深复制。

contents() 获取每个元素的子节点，返回的集合中包含文本和注释节点。

css() 获取或设置元素的style属性值。

data() 存储或返回与元素关联的指定数据值。

eq() 获取指定索引位置的元素。

① 这里的属性指的是attributes。——译者注

find()　过滤元素的子节点，只能通过标签名过滤。

hasClass()　确定元素本身是否分配了给定的类。

html()　获取或者设置元素的HTML内容。

next()　获取紧跟元素的兄弟元素。

off()/unbind()　通过名称移除一个事件处理程序。

parent()　获取元素的父元素。

prepend()　将内容插入到元素的开头。

prop()　获取或设置元素的属性[1]值。

ready()　指定一个DOM加载完成时执行的函数。

remove()　从DOM中移除元素。

removeAttr()　从元素中移除一个属性（attribute）。

removeClass()　从元素中移除一个、多个或者所有类。

removeData()　从元素中移除先前存储的数据。

repalceWith()　使用提供的新内容替换元素。

text()　获取或者设置元素中合并的文本内容。

toggleClass()　从元素中添加或者移除一个或者多个类。

triggerHandler()　执行附加给元素的某个事件的所有事件处理程序。

val()　获取或设置元素的当前值。

wrap()　使用指定的HTML结构包裹元素。

34.2　了解基本工具

AngularJS社区非常出色，还编写了一些非常不错的工具以支持AngularJS开发。接下来我们将讨论构建工具、框架以及实时交互工具。

34.3　Grunt

Grunt[2]是一个纯净的JavaScript任务运行器。开发JavaScript应用程序时它可以为你节省很多时间，这包括服务器端和客户端。它会让重复任务消失，并且还可以为你自动处理运行任务。

JavaScript社区到处都在用Grunt工具，并且已经创建了数百个插件。如果需要或者想用的插件还没有人开发，使用Grunt工具创建自己的插件也非常容易。

① 这里的属性指的是property。——译者注

② http://gruntjs.com/

安装

首先，必须确保你已经安装了NodeJS[①]。NodeJS是一个以Chrome的JavaScript运行时为基础的平台；它允许你使用JavaScript作为服务端语言编写程序。

要安装Grunt，可以使用NodeJS自带的内置npm工具：

```
$ npm install -g grunt-cli
```

> 传递-g标志可以让grunt命令运行于你计算机上的任意目录。

安装好Grunt后，还需要在应用中使用一个Grntfile来配置Grunt如何运行以及运行什么。为了使用Grunt做一些有用的事情，让我们一起在项目中创建一个Gruntfile.js。

首要的事情是，必须创建一个package.json文件来告诉Node要安装哪些东西作为依赖。

> 就像AngularJS依赖处理一样，NodeJS也有一个用于依赖管理的巧妙的方法。package.json文件将会是你编写更多NodeJS应用的助手。

要建立默认的package.json文件，可以运行生成器或者从默认的package.json中复制粘贴。由于`npm init`命令是内置的，就使用它吧：

```
$ npm init
```

这个命令会问一系列问题，比如新应用的名称、版本以及更多问题。也可以使用它设置所有默认值，例如设置应用的名称就是个不错的选择。

完成这个命令后，它会创建一个package.json文件，看起来像这样：

```
{
    "name": "myapp",
    "version": "0.0.0",
    "description": "Your myapp description",
    "main": "index.js",
    "scripts": {
        "test": "echo \" Error: no test specified\" && exit 1"
    },
    "author": "Your name",
    "license": "MIT"
}
```

可以通过再次使用npm命令将基本的grunt命令安装到package.json文件中：

```
$ npm install grunt --save-dev
```

> --save-dev标记用于将grunt作为开发依赖保存。如果想将它作为运行时依赖，可以只使用--save标记。

Grunt常用于压缩JavaScript文件，这样就可以发送尽可能小的文件给浏览器。这个过程特别

① http://nodejs.org/

有用，它可以尽可能快地加载应用，特别是在移动设备上。

这里使用Uglify插件来为我们处理这个过程。

```
$ npm install grunt-contrib-uglify --save-dev
```

非常好！现在可以使用Grunfile来配置Grunt了。要配置Grunt，只需在文本编辑器中载入Gruntfile.js文件然后添加以下代码：

```
module.exports = function(grunt) {
    // 配置
    grunt.initConfig({
        pkg: grunt.file.readJSON('package.json')
    });
    // 加载插件
    // 默认任务
};
```

为了设置一个配置，需要让Grunt载入我们想使用的所有插件的npm任务。由于这里正在使用uglify任务，因此需要让Grunt载入grunt-contrib-uglify插件任务：

```
grunt.loadNpmTasks('grunt-contrib-uglify');
```

要配置Uglify，可以在`initConfig`对象内使用`uglify`属性放入一个配置块。在这个例子中，先创建一个很小的更新配置，只设置了`src`和`dest`位置：

```
grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    uglify: {
        build: {
            src: 'src/<%= pkg.name %>.js',
            dest: 'build/<%= pkg.name %>.min.js'
        }
    }
});
```

`grunt-contrib-uglify`模块的所有可配置选项都可以在项目的README中找到（https://github.com/gruntjs/grunt-contrib-uglify）。

> 注意，使用Grunt模块时，配置文档通常都在项目的README文件中，或者README可能指向其可用的配置选项。

使用上面这个设置，Grunt会在src/目录中使用我们给package.json的名称来查找JavaScript文件。然后在这个文件上运行Uglify任务。

为了真正告诉Grunt运行这个任务，你可以简单地运行`uglify`任务：

```
$ grunt uglify
```

也可以通过声明一个带多个子任务的任务，来配置Grunt在一个任务中运行多个任务：

```
grunt.initConfig({
    // 配置
});
grunt.register('default', ['uglify']);
```

现在，你可以运行`grunt default`，然后定义在数组中的所有任务都会运行。在Grunt中

default任务具有特殊意义，对于像这样配置的default任务，可以只运行grunt命令，所有这些任务都会运行。

你可能对这个功能为什么如此有用比较好奇。在这个例子中，我们只设置了运行一个任务，但是如果使用的是CoffeeScript，你会想将所有Angular模板打包为一个独立文件，打包less CSS文件，等等，Grunt可以为你处理所有这些。

最后，Grunt最有用的特性之一是，它能够监控文件系统的变化以及在变化的文件上执行命令。

要设置监控，只需遵循与上面相同的两个步骤。

首先，要安装grunt-contrib-watch这个npm包：

```
$ npm install grunt-contrib-watch --save-dev
```

接下来，在initConfig对象中设置一个配置块：

```
grunt.initConfig({
    pkg: grunt.file.readJSON('package.json');
    //
    watch: {
        js: {
            files: 'src/**/*.js',
            tasks: ['uglify']
        }
    }
});
```

现在你可以运行grunt watch，Grunt会监控位于src/目录中的所有JavaScript文件。在任何文件发生变化时，它都会运行uglify任务。

34.4 grunt-angular-templates

默认情况下，Angular无法从本地$tempalteCache中找到模板时，会通过XHR提取模板。当XHR请求很慢，或者模板很大时，它可能会对应用的用户体验造成很大的负面影响。

你可以通过“伪造”$templateCache已经被填充的方式来避免这一延迟，这样Angular就不必从远程加载模板。可以在JavaScript中手动实现这个技巧，就像这样：

```
angular.module('myApp',[])
.run(function($templateCache) {
    $templateCache.put('home.html', 'This is the home template');
});
```

现在，当Angular需要提取名为home.html的模板时，它会在$templateCahce中找到它，而无需从服务器提取。

如果想为服务器打包应用，手动处理的步骤就会很繁琐。幸好，grunt-angular-templates这个Grunt任务可以帮我们完成。

34.4.1 安装

首先，需要安装这个Grunt任务。可以使用npm以如下方式安装它：

```
$ npm install --save-dev grunt-angular-templates
```

> 这里使用--save-dev任务会将这个Grunt任务存储到package.json文件中，这是一种很好的做法。保存依赖和版本时可以很容易地为应用程序创建新的开发环境。如果没有使用package.json文件，可以忽略这个标记，它不会造成任何损害。npm将会输出一个没有使用package.json的警告信息。

接下来，需要在Gruntfile.js中引用这个新任务，就像这样：

```
grunt.loadTasks('grunt-angular-templates');
```

现在，你可以在Grunt任务中安全地使用这个任务了。

34.4.2 用法

这个任务本身会编译一个JavaScript文件，你需要在你的index.html加载它。例如，如果让这个任务生成templates.js文件，那么就需要在index.html中载入它：

```
<script src="templates.js"></script>
```

首先，和任何其他Grunt任务一样，你需要配置它。配置模板的属性是`ngtemplates`。在这个`ngtemplates`配置块内，我们设置了一个子任务，它的名称就是我们加载的Angular模块的名称。

例如：

```
ngtemplate: {
    myApp: {}
}
```

这个代码生成的templates.js文件输出为：

```
angular.module('myApp')
.run(['$tempateCache'], function($templateCache) {
    $templateCache.put('home.html', ...);
})
```

子任务的名称`myApp`与Angular模块的名称相同，`$templateCache`会将它放到它的模板中。

而选项将会设置在这个子任务内。

34.4.3 可用选项

1. bootstrap

默认情况下，`grunt-angular-templates`会把`function($templateCache) {}`包裹到`angular.module('myApp').run(['$templateCache', __]);`内。如果你在CommonJS或者RequireJS中使用了`bootstrap`选项，那么可以改变这个配置：

```
// ...
bootstrap: function(module, exports) {
    return 'module.exports[module]=' + script + ';';
}
```

2. concat

concat是concat定义内的目标名称，这是你要添加编译后的模板路径的地方。

3. htmlmin

正如可以压缩CSS和JavaScript文件一样，还可以使用一个叫做（意料之中）htmlmin的工具来压缩HTML。grunt-angular-templates能够很好地和htmlmin一起工作，此外，它甚至还允许压缩模板内的HTML。

你可以在配置内为htmlmin设置选项，就像这样：

```
ngtemplates: {
    myApp: {
        options: {
            htmlmin: {
                collapseBooleanAttributes: true,
                collapseWhitespace: true,
                removeAttibuteQuotes: true,
                removeEmptyAttributes: true,
                removeRedundantAttributes: true,
                removeScriptTypeAttributes: true,
                removeStyleLinkTypeAttributes: true
            }
        }
    }
}
```

4. module

angular.module的名称，模板缓存会将这个名称注册为模板要缓存的模块。

```
ngtemplate: {
    myApp: {
        options: {
            module: 'myBestApp'
        }
    }
}
```

因此将会看到以下方式的模板设置：

```
angular.module('myBestApp')
    .run(['$templateCache'],
    function($templateCache) {
        // ...
    });
```

5. prefix

我们可以为所有的模板URL设置一个前缀。例如，如果想使用绝对URL来存取来自不同目录中绝对位置的模板，要像这样设置前缀：

```
ngtemplate: {
    app: {
        options: {
            prefix: '/public'
        }
    }
}
```

6. source

可以将source选项设为一个函数，它会在模板被编译之前，源文件被压缩之后调用，因此可以自定义模板文件输出。例如，可以给源文件添加一个标准的头部。

这个函数会使用这些选项调用。

- source：压缩后的模板资源。
- path：模板文件路径。
- options：任务选项对象。

7. standalone

这个布尔值选项标记用于告诉Grunt任务模板，它到底是独立的，还是现有模块的一部分，比如myApp。大多情况下，这个选项都应该设置为false（这也是它的默认值）。

8. url

设置url选项时会重写模板的$templateCache URL。通常，特殊情况下才使用这个选项；设置cwd和src可以让模板用于XHR和$templateCache。

34.4.4　用法

grunt-angular-templates的作者提供了许多选项，告诉我们可以如何使用这个任务。

1. concat

使用该任务的最简单方式就在concat任务内。它的任务是负责压缩concat任务内的位置。

```
concat: {
    app: {
        src: ['*.js', '<%= ngtemplates.app.dest %>'],
        dest: ['app.js']
    }
}
```

现在模板将被附加到app.js文件的尾部。

2. usemin

grunt-usemin是一个压缩和合并内联请求JavaScript文件的任务，可以在产品中压缩和合并源文件，但是在开发中继续使用未压缩的依赖。例如：

```
<!-- build:js module.js -->
<script src="scripts/app.js"></script>
<script src="scripts/controllers.js"></script>
<!-- endbuild -->
```

文件会被压缩为module.js。然后可以将这个文件作为目标文件附加给模板。

```
ngtemplates: {
    app: {
        src: 'templates/*.html',
        dest: 'template.js',
        options: {
            concat: 'module.js'
```

```
        }
    }
}
```

3. dest

最后，还可以通过指定一个目标位置来正常地生成templates.js，而不是简单地分配一个`dest:`键作为文件路径插入另一个文件。

```
ngtemplates: {
    app: {
        src: 'templates/*.html',
        dest: 'template.js'
    }
}
```

34.5 Lineman

Lineman是一个构建工具，它允许我们主要关注构建客户机（或者客户端）Web应用。它混合了很多令人难以置信的功能，让客户端webapp开发变得有趣和容易。

Lineman由社区构建和维护，它使得前端webapp开发多产，并且可维护、可管理。

本书一直都在单个index.html文件中开发客户端应用程序，我们一直使用浏览器加载该应用。Lineman使用了不同的方式，它凭借本地服务器服务于应用程序。

使用本地服务器为文件提供服务时，Lineman可以提供那些不能在静态文件中使用的特殊功能。包括：

- 保存文件时编译CoffeeScript①文件为JavaScript；
- 运行Less②和Sass③预处理器生成CSS；
- 提供后端替代工具，因此后端服务器可有可无；
- 预编译JavaScript模板；
- 为后端服务器代理XHR请求；
- 让测试更容易和有趣。

Lineman明确不会处理任何后端服务器（尽管它提供了一种调用后端的方式，正如我们将会看到的）。它重点关注构建可以编译、压缩和部署为静态Web应用的AngularJS应用。

要使用Lineman，需要确保安装了NodeJS④；它自带了用于打包的npm工具。为了使用Lineman，可以使用npm全局安装它。

```
$ npm install -g lineman
```

虽然我们将使用Lineman运行项目，但这里还没使用打包生成器。相反，这里将会使用由David Mosher创建的AngularJS模板。

① http://coffeescript.org/
② http://lesscss.org/
③ http://sass-lang.com/
④ http://nodejs.org/

```
$ git clone https://github.com/davemo/lineman-angular-template my-app
```

克隆好这个模板之后（使用Git），便于操作，可以使用npm再次安装Lineman需要的依赖：

```
$ cd my-app && npm install -d
```

依赖安装好之后，就可以开始开发应用了。接下来我们会编写这个运行测试以及服务器的应用。

为了运行应用，还需要在my-app目录中启动Lineman工具。

```
$ lineman run
```

至此，我们的应用会运行在浏览器的http://localhost:8000中，如图34-1所示。



图34-1　运行Lineman

正如你看到的那样，Angular有几个生成应用的模板。可以看到这个目录结构还包含一些其他目录。

- **app**：包含应用程序文件。
  - **css**：CSS文件（Less或者CSS文件）。
  - **img**：图片文件。
  - **js**：Angular应用。
  - **pages**：要编译的HTML模板。
  - **templates**：Angular模板。
- **config**：Lineman特定的配置文件。
- **doc**：应用文档目录。
- **dist**：一个生成的用于构建应用的目录。
- **generated**：使用lineman run运行应用生成的目录。
- **spec**：所有非端到端的说明文档。
- **spec-e2e**：分度器说明文档。
- **tasks**：所有自定义的Lineman任务都应该存放在这里。
- **vendor**：包含所有第三方CSS、JavaScript和图片文件。
- **Gruntfile.js**：提供`lineman`的Gruntfile。
- **package.json**：应用程序定制，定义依赖和其他元数据。

Lineman提供了一个可以快速高效编写Web应用的结构。

34.6 Bower

Bower是Web开发中的一个前端文件包管理器。类似于Node模块的npm包管理器，它允许开发者为服务器编写可共享的模块。Bower为Web组件提供了类似的功能。

它凭借一个通用的、中性且易用的接口为依赖问题提供了一个解决方案。它是基于Git运行的，并且包是未知的。它还支持其他传送类型，比如requireJS、AMD，等等。

34.6.1 安装

安装很简单：只需使用包管理器npm安装bower即可：

```
$ npm install -g bower
```

bower依赖于Git、Node和npm。

然后，可以通过输入help命令来确认它是否安装成功：

```
$ bower help
```

如果输出显示界面如图34-2所示，表示可以使用了。

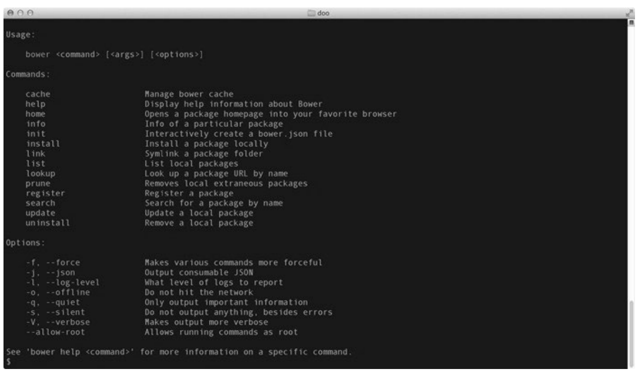


图34-2 Bower帮助视图

34.6.2 Bower简介

尽管这里只会涵盖一些简短的简介，但是鼓励到Bower主页：bower.io[①]进行更多的探索。

对于Web应用，你可能想要与其他开发人员共享源代码或者部署到其他开发机器上。与适用于npm的package.json类似，可以使用一个bower.json文件存储前端依赖。

① http://bower.io/

为了开始使用bower.json，可以使用Bower提供的`init`命令。我们应该在项目的根目录执行它：

```
$bower init
```

这条命令会启动一个设置向导，它问一些关于新程序包的问题。回答完以后，会在当前目录生成一个新的bower.json文件。

34.6.3　配置Bower

Bower自带了健全的默认配置，但它也是高度可配置的。你可以配置安装程序包的目录，并注册哪个目录用于安装组件。

> https://docs.google.com/document/d/1APq7oA9tNao1UYWyOm8dKqlRP2blVkROYLZ2fLIjtWc/edit#heading=h.4pzytc1f9j8k上可以看到更多的Bower配置文档。推荐你参考这份文档了解更多详细配置信息。

尽管深入了解`bower`的配置信息不在本章范围之内，但我们将会看到两个最常见的修改配置项（基于我们自己的经验）。

要配置Bower，可以编辑.bowerrc文件，传递配置参数，或者设置环境变量。还可以将.bowerrc文件放在不同的地方：

- 项目当前工作目录中；
- 目录树的任意子目录中；
- 当前用户的主目录中；
- 全局的Bower目录中。

.bowerrc文件包含一个适用于配置的JSON对象。比如，要改变颜色配置，.bowerrc文件应该包含：

```
{
    "color": false
}
```

为了简单起见，这里我们将.bowerrc文件放在项目的根目录中。如果不存在，推荐在项目的根目录中创建它：

```
$ echo "{}" > .bowerrc
```

cwd　`cwd`配置变量表示应该从哪个目录运行Bower。所有其他路径都应该直接相对于这个目录。

```
{
    "cwd": "app"
}
```

directory　`directory`配置变量表示安装的组件应该保存在哪个路径中。默认为bower_components。这依赖于如何创建应用，可以修改这一配置以适应不同的目录结构：

```
{
    "directory": "app/components"
}
```

34.6.4 搜索程序包

为了找到程序包用于安装，Bower包含了一个搜索命令用于搜索注册的索引：

```
## Searching for bootstrap-sass
$ bower search bootstrap-sass
```

34.6.5 安装程序包

安装程序包同样很简单。如果有一个现有的bower.json文件，可以简单地运行安装命令。它会拉取并安装前端依赖到Bower目录中：

```
$ bower install
```

你可以通过在文件上显示调用安装命令的方式，安装程序包到本地。也可以安装指定版本的程序包，甚至为程序包的安装设置一个*别名*。

```
# Install a local or
# default remote version of a package
$ bower install <package>
# Install a specific version of a package
$ bower install <package>#<version>
# Alias install a package
$ bower install name=<package>#<version>
# For instance
$ bower install bootstrap=bootstrap-sass
```

bower.json文件可以存储多个类型的依赖：要么是运行时的依赖（比如Angular或者jQuery），或者是开发过程中需要的依赖（比如karma或者Bootstrap-sass）。

```
# Install a run-time dependency
$ bower install angular-route --save
# Install a dev dependency
$ bower install bootstrap-sass --save-dev
```

如果将bower.json文件的内容打印出来，将会看到使用新安装的依赖更新后的内容：

```
$ cat bower.json
{
    "name": "myApp",
    "version": "0.0.1",
    "authors": [
        "Ari Lerner <ari@fullstack.io>"
    ],
    "license": "MIT",
    "dependencies": {
        "angular-route": "~1.2.13"
    },
    "devDependencies": {
        "bootstrapp-sass": "~3.0.0"
    }
}
```


34.6.6 使用程序包

现在程序包已经安装好了，我们可以通过在HTML源代码中使用`script`标记的方式引入这些

程序包，就像引入本地目录中的任何其他脚本一样。

```
<script
src="/bower_components/angular/angular.js">
</script>
```

34.6.7　移除程序包

使用Bower移除程序包也是可能的。可以在Bower目录手动删除文件，或者运行`uninstall`命令。

这个`uninstall`命令允许我们使用`--save`和`--save-dev`标记映射bower.json文件的变化。

```
# Remove a dependency
$ bower uninstall --save-dev angular-route
# Remove a devDependency
$ bower uninstall --save-dev bootstrap-sass
```

34.7　Yeoman

Yeoman[①]是本章前面讨论过的一个工具的集合：

- Yeoman；
- Grunt；
- Bower。

Yeoman本身是一个脚手架工具，通过配置Grunt配置，构建应用程序工作空间和管理工作流程，以及帮助我们构建新的应用程序，不管我们构建的应用程序类型。

在编写本书时，大约有300个社区编写的生成器可用，这些生成器已经建立了很多不同类型的项目，从Angular[②]站点到Backbone.js[③]站点，甚至还包括Python flask[④]项目。

Grunt被设定为构建工具，而Bower用来处理依赖管理。

34.7.1　安装

安装Yeoman很简单。首先，需要确保安装了Node.js[⑤]和Git[⑥]。某些生成器可能还需要安装Ruby[⑦]和Compass[⑧]。

安装好这些依赖之后，可以使用`npm`安装Yeoman：

```
$ npm install -g yo
```

① http://yeoman.io/
② https://github.com/yeoman/generator-angular
③ https://github.com/yeoman/generator-backbone
④ https://github.com/romainberger/yeoman-flask
⑤ http://nodejs.org/
⑥ http://git-scm.com/
⑦ https://www.ruby-lang.org/
⑧ http://compass-style.org/

安装Yeoman时会自动安装**Grunt**和**Bower**。

接下来，为了使用Yeoman，还需要安装生成器（Yeoman本身不带生成器）。

我们来安装Angular生成器：

```
$ npm install -g generator-angular
```

要搜索社区中所有可用的生成器，可以参考http://yeoman.io/communitygenerators.html界面。

34.7.2 用法

使用Yeoman工作流也很容易。首先最重要的是，要创建一个可以进行工作的目录。Yeoman并不会为我们创建工作目录；相反，它会假定我们正在使用的目录就是存放应用的目录。

```
$ mkdir myapp && cd $_
```

我们将会在这个目录内运行生成器架构项目。在这个例子中，我们使用generator-angular这个Angular生成器，如图34-3所示。

```
$ yo angular
```

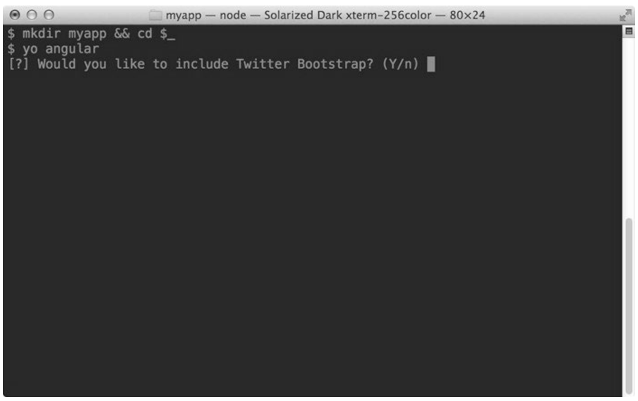


图34-3 Yeoman安装

Yeoman会问一些问题，然后创建应用程序。在这些步骤中，它会调用npm install和bower install确保所有的依赖已经存在，这样便可以立即进行开发。

我们将使用grunt命令开始我们的开发过程。

```
$ grunt server
```

grunt server命令会启动一个本地服务器在本地为应用提供服务。当我们在工作空间中保存文件时它会使用livereload[1]自动重载浏览器。

它为我们构建的这个目录有一个坚实的结构，用于部署容易扩展的Angular应用，如图34-4所示。

① http://livereload.com/

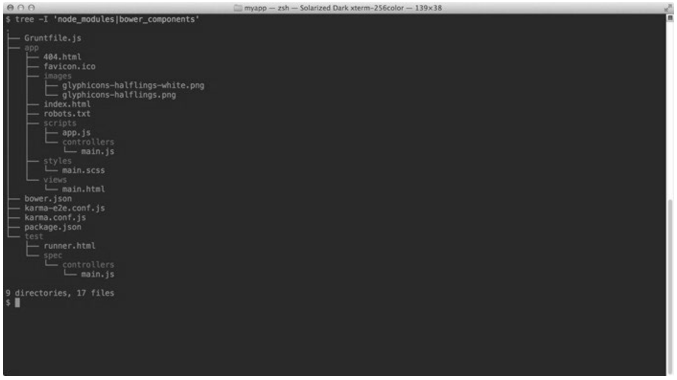


图34-4　Yeoman生成的目录结构

Yeoman创建的目录结构构建了app/和test/目录。在app目录内，将会用来构建Angular应用和存放视图、样式和应用程序的其他部分。

当我们想要创建一个控制器时，需要添加一个带有描述名称的文件到控制器目录。然后还需要确保在index.html中引入它作为文件加载。

例如，添加一个仪表盘控制器时，将会创建app/scripts/controllers/dashboard.js和`DashboardController`定义：

```
'use strict';

// 在 app/scripts/controllers/dashboard.js中
angular.module('myappApp')
    .controller('DashboardController', function($scope) {
    });
```

为了引入这个控制器，需要让应用程序在index.html中加载这个文件。还需要确保在应用内的构建注释中引入它，这样htmlmin任务才会在压缩HTML时引入它。

```
<!-- build.js({.tmp, app}) scripts/scripts.js -->
<script src="scripts/app.js"></script>
<script src="scripts/controllers/main.js"></script>
<script src="scripts/controllers/dashboard.js"></script>
<!-- endbuild -->
```

这样就可以在应用中使用这个控制器了。这个过程可用于将在应用中使用的各类Angular组件（例如服务、过滤器和指令）。

如果将应用分离为多个组件（强烈推荐）作为应用的依赖，需要确保在上面的app.js文件之前引入这些组件。例如，如果遵循多模块模式将会为每个组件生成一个新模块：

```
// 在app/scripts/services/api.js中
angular.module('myApp.services', [])
    .factory('ApiService', function() {
        return {};
    });
```

然后设置这些模块作为应用的依赖：

```
// 在app.js中
angular.module('myApp', ['myApp.services']);
```

最后在HTML中还需要在引入app.js之前引入这些服务。

此外，Angular生成器还自带了一些有用的生成器，让构建Angular应用的过程变得更容易。

34.7.3 创建路由

要创建一个包含控制器的路由以适应控制器测试，在HTML中引入<script>标签并为路由创建视图，需要在终端中运行以下命令，如图34-5所示。

```
$ yo angular:route home
```



图34-5 创建一个新路由

34.7.4 创建控制器

为了创建一个简单的控制器以适应测试，可以在终端中使用生成器：

```
$ yo angular:controller user
```

34.7.5 创建自定义指令

为了创建指令适应测试，还可以使用如下命令创建指令：

```
$ yo angular:directive tabPanel
```

34.7.6 创建自定义过滤器

还可以在应用中创建自定义过滤器以适应测试。为了做到这一点，可以使用如下生成器：

```
$ yo angular:filter capitalize
```

34

34.7.7 创建视图

为了生成一个简单的视图，也可以使用Angular生成器命令：

```
$ yo angular:view dashboard
```

34.7.8 创建服务

此外，也可以使用生成器创建服务。下面我们以不同格式创建了一个服务以适应测试。

```
$ yo angular:service api
$ yo angular:factory api
$ yo angular:provider api
$ yo angular:value api
$ yo angular:constant api
```

34.7.9 创建装饰器

Angular生成器还可以在其他服务之上创建装饰器。只需在终端执行这个命令：

```
$ yo angular:decorator api
```

34.8 配置 Angular 生成器

对于前面提到的生成器（包括主生成器），我们都可以传递选项以自定义的方式配置脚本。

34.8.1 CoffeeScript

如果想要生成CoffeeScript文件而不是JavaScript文件，通过传递`--coffee`选项可以很容易做到：

```
$ yo angular:controller user --coffee
```

34.8.2 安全压缩

尽管这不是必要的（因为Yeoman生成器包含`ngMin`），仍然可以使用`--minsafe`标记，让生成器在生成的文件中加入依赖注入声明：

```
$ yo angular:controller user session--minsafe
```

34.8.3 跳过索引

默认情况下，前面提到的所有生成器都会在index.html中添加适当的文件进行加载。我们也可以让生成器在index.html中不引入脚本。

> 你可能希望跳过添加文件到主页中，比如构建第三方插件。

```
$ yo angular:factory session --skip-add
```

34.9 测试应用

Yeoman的Angular生成器最好的特性之一便是，它允许我们在开发应用时对应用程序进行无缝测试。

这个生成器打包了一个测试命令，它会在我们在应用中保存文件时运行。这个打包让测试程序很容易集成到工作流程中。

要运行测试而无需监控文件（比如，运行一次），可以使用这条命令：

```
$ grunt test
```

这条命令运行一次之后就会退出。推荐对这个工作流程做出两个修改从而在应用中引入自动化测试。

首先，在应用程序的根目录打开Gruntfile.js文件，找到Karma任务。然后，将选项singleRun从true改为false：

```
        // ...
    ]
},
karma: {
    unit: {
        configFile: 'karma.conf.js',
        singleRun: false // 将这个选项改为false
    }
},
cdnify: {
// ...
```

其次，打开karma.conf.js文件，将autoWatch选项从false改为true。

现在，运行grunt test时，不再是运行一次后退出，这个任务将会一直开着，同时监控文件。只要改变文件并保存，这个测试任务就会再次运行。

34.10 打包应用

在开发好应用程序之后，我们想要建立一个发布版本的应用。创建发布版本的应用时，要包含所有压缩后的JavaScript和HTML文件，还要打包视图，预处理CSS，等等。

要运行构建任务，可以简单地运行grunt build命令：

```
$ grunt build
```

这会花一些时间运行完整的生成器。当任务完成时，在应用的根目录会得到一个dist/文件夹。这个文件夹包含所有适用于产品部署的文件。

可以将这个文件夹上传到服务器中，或者部署到服务器中让应用程序能够被用户访问。

34.11 打包模板

要使应用程序显示得更快，并且无需依靠服务器传输模板，我们可以使用的方法是将模板文件转换为JavaScript文件。

使用Angular的templateCache时，我们会把模板包含到JavaScript文件中。例如，使用XHR替代Angular提取HTML时：

```
<div class="hero-unit">
    <h1>'Allo, 'Allo!</h1>
    <p>You now have</p>
    <ul>
        <li ng-repeat="thing in awesomeThings">{{thing}}</li>
    </ul>
    <p>installed.</p>
    <h3>Enjoy coding! - Yeoman</h3>
</div>
```

你可以打包它们到一个JavaScript中，然后分离这个JavaScript文件，就像这样：

```
angular.module('myApp')
    .run(['$templateCache', function($templateCache) {
        $templateCache.put('views/main.html',
        "<div class=\"hero-unit\">\n" +
        "  <h1>'Allo, 'Allo!</h1>\n" +
        "  <p>You now have</p>\n" +
        "  <ul>\n" +
        "    <li ng-repeat=\"thing in awesomeThings\">{{thing}}</li>\n" +
        "  </ul>\n" +
        "  <p>installed.</p>\n" +
        "  <h3>Enjoy coding! - Yeoman</h3>\n" +
        "</div>\n"
    );
}]);
```

要完成这个设置，需要修改Gruntfile.js引入新任务定义，来使用新的npm包grunt-angular-templates。

首先，要安装这个包：

```
$ npm install --save-dev grunt-angular-templates
```

接下来，修改Gruntfile.js引入ngtemplates任务。

```
// ...
    }
},
ngtemplates: {
    myappApp: {
        cwd: '<%= yeoman.app %>',
        src: 'views/**/*.html',
        dest: '<%= yeoman.app %>/scripts/templates.js'
    }
},
// 将不处理的文件放在其他任务中
copy: {
// ...
```

这一修改仅在应用目录中创建了一个新文件,这个文件将会包含作为JavaScript加载的模板文件。

这里还需要确保在构建过程中运行这个任务。幸好，将这个任务添加到构建过程中很容易。只需找到：grunt.registerTash('build', [所在行，然后确保将ngTemplates添加到这个任务数组的concat任务后面：

```
grunt.registerTask('build', [
    // ...
    'concat',
    // ...
```

```
    'cssmin',
    'ngtemplates',
    'uglify',
    'rev',
    'usemin'
]);
```

最后，还需要确保在app/index.html中引入scripts/app.js文件之后，引入这个templates.js文件：

```
<!-- build.js({.tmp, app}) scripts/scripts.js -->
<script src="scripts/app.js"></script>
<script src="scripts/controllers/main.js"></script>
<script src="scripts/templates.js"></script>
<!-- endbuild -->
```

至此，当构建应用时，模板文件将会和应用程序的其他部分一起打包。

注意，在开发应用程序时，如果在缓存中没有找到模板，它会自动从服务器加载，因此，如果有需要，在开发的过程中可以安全地删除app/scripts/template.js文件。

如果这个文件已经存在了，视图会使用缓存的文件而不会重新加载；它会认为有可用的模板。

第 35 章

总　　结

现在你已经掌握了相关的知识和实践，可以熟练地使用Angular构建强大的应用了。感谢你与我们一起探索这个框架。AngularJS社区非常庞大并且还在不断成长，让我们一起促进它的发展壮大吧！我们期待看到你的应用程序。

亚马逊读者评论

“在此之前，我为了学习AngularJS总是在网上搜索阅读别人写的博客，但头脑中始终没法建立一个大局观。这本书内容真的很全很全，而且组织得当，讲解通俗，让我对AngularJS有了全面系统的理解。”

“我看过好几本AngularJS图书了，只有本书全面、详尽，循循善诱，通俗易懂，其他任何一本都无法与它比肩。”

“ 显然，作者Ari Lerner不仅是一位卓越的软件工程师，而且还是一位天才的老师和作者。他把AngularJS的各种功能讲了一个遍，很多抽象的概念都讲得透彻易懂。”

AngularJS是Google开发的下一代产业级Web应用框架，最早发布于2009年。随着全球众多Web项目（包括SPA，即单页应用）的竞相采用，AngularJS进入了成熟稳定期，是学习和研究下一代Web开发的首选框架。

说到学习AngularJS，相信你早已厌倦了上网搜索、断续阅读的低效方式。本书堪称AngularJS领域的里程碑式著作，它以相当的篇幅涵盖了关于AngularJS的几乎所有内容，既是一部权威教程，又是一部参考指南。对于没有经验的人，本书平实、通俗的讲解，递进、严密的组织，可以让人毫无压力地登堂入室，迅速领悟新一代Web应用开发的精髓。如果你有相关经验，那本书对AngularJS概念和技术细节的全面剖析，以及引人入胜、切中肯綮的讲解，将帮助你彻底掌握这个框架，在自己职业技术修炼之路上更进一步。

本书是资深全栈工程师的代表性著作，由拥有丰富经验的国内AngularJS技术专家执笔翻译，通俗易懂、全面深入，是学习AngularJS不可错过的经典之作。无论是出于工作需要，还是好奇心的驱使，只要你想彻底理解AngularJS，本书都会让你感到满意。

本书将涵盖AngularJS的如下概念。

- 双向数据绑定
- 依赖注入
- 作用域
- 控制器
- 路由
- 客户端模板
- 服务
- 通过XHR实现动态内容
- 测试
- 过滤器
- 定制表单验证
- 深度测试
- 定制指令
- 专业工具
- 对IE的支持

图灵社区：iTuring.cn
热线：(010)51095186转600
分类建议 计算机/网站开发/AngularJS

ISBN 978-7-115-36647-4
ISBN 978-7-115-36647-4
定价：99.00元

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks